Containers' Privacy and Data Protection via Runtime Scanning Methods

Francisco Rojo(✉) and Lei Pan

Deakin University, Geelong, VIC 3220, Australia
{frojorosales,l.pan}@deakin.edu.au

Abstract. Docker containers' privacy and data protection is a critical issue. Unfortunately, existing works overlook runtime scanning methods. This paper proposes a novel lightweight and rapid scanning model under a framework covering assertion techniques during the container's runtime, defined as *vulnerability scanning framework* VSF. Our framework includes identifying vulnerability, scanning security exposures, conduct analysis, and call-back notifications to the requestor asynchronously. In addition, the proposed scanning model is compared against other tools of similar and complementary objectives. The framework is modeled using *nmap* scripting engine NSE for its active scanning building block. It applies network port scanning and security assertion techniques to rapidly discover security vulnerabilities in a running Docker container environment for a proactive testing approach as a security engine. Also, providing an active trust model developed for Docker containers whether containers are *black-listed* or *grey-listed*. It was developed over a framework for DevSecOps environments and DevOps teams as the persona on its adoption. The empirical case studies demonstrate the capability of our scanning model, including standalone, CI/CD pipelines, and security containerized environment. The case studies revealed no tangible difference in the performance but the flexibility driven by the modeled architecture. The experiments presented a velocity of $1.15\frac{scans}{sec}$. However, the execution time is directly proportional to the complexity of the vulnerability on the Docker ecosystem and its related attack vector complexity. Its core capability resides on the artifacts developed as part of the Art per relevant CVE via *nmap* NSE scripts.

Keywords: DevSecOps · DevOps · Containers · Docker · Containers security · Docker security · Containers vulnerability scanner · Containers vulnerability assertion · Vulnerability scan

1 Introduction

Information Technology (IT) ecosystems that are generally considered secure with a full spectrum of security measures can be exposed to vulnerabilities natively available in container environments, on either: the container host, guest

W. Xiang et al. (Eds.): BROADNETS 2021, LNICST 413, pp. 37–56, 2022.
https://doi.org/10.1007/978-3-030-93479-8_3

daemon, and the image. Large Enterprise, Banking, Government, Telco, Public Cloud, Entertainment amongst key players in today's economy, are widely adopting containers platforms as an emerging technology due to its native benefits; driving microservices architectures adoption in a vast range of organisations such as Amazon, Twitter amongst a few [13]. Some key drivers in the emergence of containerised environments include simplicity, flexibility derived from a microservices architecture, shared compute underlying options, and easy adoption requirements.

DevOps teams are surging as digital technology enablers across organisations and using containers as a crucial component. The organisational size and business requirements drive the container's orchestration needs, where advanced environments will consume containers via a clustered orchestration layer, such as Kubernetes. Some other players will deploy isolated container hosts to meet their needs on a smaller scale. Despite the approach taken, the key issue remains the same, caused by the existing vulnerabilities in the runtime containers' abstraction layer.

DevOps environments tend to adopt conventional security measures and passive image scanning, where related work presents novel active and passive security methods that aim to provide a secure and trusted environment. The Docker environments rely on Docker Hub as the sole repository of public Docker images, including non-official nor verified images available. Once a Docker image is loaded into a Docker Host, Docker does not provide a method to prove the image authenticity in runtime. However, Guo et al. [5] proposes a PKI under a container attestation service. The ability to deploy unverified Docker images presents a security risk in any Docker environment, given published images are public, uncontrolled, nor digitally signed.

The DevSecOps paradigm arises on protecting a DevOps tool-chain if the tool-chain runs on containers or uses containers as an underpinning technology. How can the containers be trusted? This paper proposes to black-list containers that are directly pulled from the Docker Hub until verified under the proposed *vulnerability scanning framework* VSF. It grey-lists the relevant containers launched in runtime with active software scanning techniques. It allows validating vulnerabilities present in the running container like CVE (Common Vulnerabilities and Exposures) and consequently trusts the container in the containers' network environment.

The experiments were conducted on Amazon AWS EC2 compute running CentOS 7 instances. The case studies are considered typical scenarios for DevOps teams; DevOps teams are considered the key stakeholder group for adopting the proposed work. VSF's key features aligns with the flexibility sought in DevOps environments for container runtime vulnerability scanning.

Our research work found that: **(a) runtime assertion testing for vulnerability scanning in Docker environments** is an active technique to mitigate native security risks associated with the ability of Docker to deploy images that are not verified nor signed directly from Docker Hub. **(b)** A **trust model** can be leveraged via the assertion testing with using an accuracy factor to determine

whether the *grey-listed/black-listed* classification for a Docker container is certain against some conditions. **(c)** Any active Docker vulnerability scanning techniques can be adopted by DevOps teams via a **vulnerability scanning framework** as needed in DevSecOps environments.

In the following section, we present the *related work*, where we cover the classification of the tools and approaches used to date in security-relevant to containers environments. Next, we present our *proposed framework* (VSF), some experiments, and analysis. The experiments cover the *scanning engine* selection and the case studies used to demonstrate VSF. Later on, the *results* are presented to include a summary of the finding of the case studies plus our *recommendations*; finally, our *conclusions* are presented with future work.

2 Related Work

Container environments have been rapidly adopted in the industry, especially in DevOps teams, for accelerating development and its lightweight release cycle [10]. However, this approach increases surface attacks and further security exposures. Evidence of this adoption relates to Docker Hub's recording over four million images in the Docker Hub by March 2021, with an increase of 77% in three years, when compared to 2018 as per Martin et al. [10]. The key attributes of container adoption in DevOps environments include: an abstraction level in computing architecture, optimization of computing resources, and segmentation provided at the service level in a micro-services architecture [13].

Applications and services are transformed with the adoption of containers as seen in DevOps environments [10]; from edge computing to adaptive applications, when conventional security does not address the attack surface in an agile manner, a lightweight runtime scanner is required, as detailed by Merino et al. [1], as evidence on security concerns around the utilisation of containers technologies.

Containers, when compared to its predecessor technology *virtualisation* (virtual machines - VMs), drive abstraction into a micro-services domain but running on a Linux baseline kernel host. It opens up flexibility but implies fewer controls in a non-specific purpose kernel [15]. Containers have not been designed with a robust security framework, instead conventional Linux hardening and configuration options are available. Consequently, the are problems with security risks associated with the utilisation of the containers technology as applications can have direct access to the host kernel; thus, an attacker can reach the host environment from the container layer [15]. Merino et al. [1] exemplifies the need for a runtime scanning method.

We classify the related work in *hardware-software based*, or *active-passive*; being *hardware-based* dependant on compute hardware; *software-based* dependant on a software defined approach; or as *active* if it interacts with container in runtime; or *passive* when interacts with components prior attestation of containers.

Hardware-based security techniques for containers include the following: Schwarz and Lipp [12] demonstration on how side-channel attack vectors can target Intel SGX (Software Guard Extensions) chipset and how to protect it via the deployment of an enclave which proxies communications to an encrypted section of DRAM on any computing; containers can still be targeted. Guo et al. [5] developed a trust model based on remote attestation techniques for containers via vTMP (Virtual Trust Platform Modules) requiring modification of the host and container image. These use cases covered hardware-based protection on general SGX computes or attestation via vTMP, respectively, affecting container environments. In addition, Guo et al. [5] uses a PKI (Public Key Infrastructure) model during the attestation of containers. The PKI is positioned between the Host and Containers as a trust model, where the root certificate is self-signed, and certificate exchange occurs. Their method represents an active hardware technique to provide data protection with enforcement of a PKI within the container's ecosystem.

Software-based security deployments are used to protect either host, container, application, and permutations of these, including industry container hardening techniques, container isolation, vulnerabilities patching via container's lightweight images upgrades, modifications in the kernel or container images with a secure Linux load, with image vulnerability scanning occurring before the container's attestation [2,5]. Sultan et al. [13] excluded from their work the orchestration layer security and mentioned the decentralized attestation via blockchain as needing further development. Furthermore, Xu et al. [14] uses blockchain to decentralize the container image trust or other data types but developing an image trust model. Kong et al. [7] presents a secure containers' deployment method using genetic algorithms via Secure Container Deployment Strategy (SecCDS). Li et al. [9] applied a DDoS (Denial of Service) mitigation mechanisms for low rate DDoS attacks over a simulation, demonstrating that the isolation of affected containers in an environment improves the quality of service of the model via *white-listing* requests into the containers environment.

On the other hand, broader security exposure analysis reveals multi-dimensional exposures in Docker as covered by Martin et al. [10]; it highlights the importance of runtime scanning techniques. Berkovich [2] argued that scanning Docker container images as binaries are a critical security activity in DevOps environments, such as CI/CD pipelines. Yasrab [15] described the issues on the Docker container level due to the large number of sensitive services running, including the container service, application, and Host OS. A *vulnerability assessment framework* is presented by Mostajeran et al. [11] which includes three key components related to containers on their work: (1) configuration, (2) images, (3) deployed services.

A Docker Thread Detection Framework acting as a software-based system is presented by Huang et al. [6]; the framework analyses the Docker image and the container's running IP/DNS requests. Similar to IP addressing scanning, other industry players target development environments with tools such as *Kali-Linux* to target containers environments. However, this active testing is related to legacy scanning methods.

Passive scanning techniques relate to those that do not actively interact with the security exposure or presence of the condition that defines the attack vector in runtime. Guo et al. [5] that enables a trusted environment incurring in modification of the kernel and container's attestation service via a state challenge protocol. As per Kwon et al. [8] by enabling a Docker Image Vulnerability Diagnostic System (DIVDS) for containers. As well as, Berkovich et al. [2] running a container's image vulnerability scanning tool known as *Ultimate Benchmark for Container Image Scanning* (UBCIS).

On the other hand, **Active scanning techniques** relate to those which actively interact with the Docker Host and/or Docker containers in runtime to detect a condition that defines the attack vector, including work such as Mostajeran et al. [11] with their *vulnerability assessment framework* that presents a runtime fixed container security benchmark tool as a risk assessment tool. Merino et al. [1] described in its managed container layers: application, namespace, control groups, amongst others to detect containers anomalies in runtime. Alternatively, it is the potential to detect co-resident containers security exposures, according to Gao et al. [3]. Kong et al. [7] used a genetic algorithm defence system, or Huang et al. [6] which uses their *Docker thread detection framework* to complete hardware checks against computing resources and network port scanning.

Table 1 lists key comparison features in related work highlighting the *defence* as a key attribute being developed, followed by the *scanning*, *attack* and *runtime* capabilities.

Table 1. Related work comparison

Related work/Capability	Scanning	Defence	Attack	Runtime
Two-stage defense approach [3]	No	Yes	No	Yes
DIVDS [8]	Yes	No	No	No
Docker thread detection framework [6]	Yes	Yes	No	No
SecCDS [7]	No	Yes	No	No
Security assessment framework [11]	Yes	No	No	Yes
UBCIS [2]	Yes	No	No	No
Managed container framework [1]	No	Yes	No	No
Container state attestation [5]	No	Yes	No	Yes

Amongst the overall security approach discussed earlier and summarised in Table 1, the **scanning engine** is a pivotal component. Key industry players in cloud containerization as Google have developed tools such as *tsunami* [4] which can be used to develop vulnerability scanning. In our experiments section, we will compare and select a scanning engine for our proposed framework.

This work focuses on developing a lightweight security framework encompassing runtime security vulnerability scanning of the container service abstraction layer within a Docker environment as an active software-defined approach.

In particular, using the CVE disclosed vulnerabilities against Docker to enable DevSecOps practices. The framework is to be used in line with the operations of the DevSecOps environment for detecting possible attack surfaces exposed by the Docker Host or Docker Images in runtime as an active scanner. Also, it assumes that DevOps teams know the CVEs to test using VSF on the Docker environment. This method is not described in related literature, as it conducts software assertion tests of the relevant CVEs into the Docker Host or specific Docker containers in runtime.

3 Security Framework for Containers Environments

The runtime *vulnerability scanning framework* VSF for Docker containers pinnacles its capability in the active assertion testing against a Docker runtime environment in a lightweight manner. This security framework has been defined modularly to aggregate active software scanning. The active approach aims to detect existing vulnerabilities in a running Docker container environment as part of an Incident Response Procedure, Proactive testing practices, and the state of the Art container practice due to the atomic nature of the container's environment of short lifespan; and with an increased level of difficulty in its tracing and tracking capabilities once the containers are destroyed.

Initially, the positioning of the security framework is defined in the container's abstraction layer and identified as the Docker Engine; the application runs as a container via the *containerd* daemon. Thus, to complete the vulnerability assertion testing, VSF provides a binary response: *true-positive* if the container is vulnerable, or *true-negative* when the container is not vulnerable against the CVE. Hence, the container can be *grey-listed*.

Figure 1 shows a representation of VSF and its modules, including `core`, `fetch`, `runner`, `callback`, and `connection`. These modules represent specific functions underpinned by *nmap* to complete the scanning function. The framework include the following five components:

- `Core`: The core engine for the scanner like *nmap*.
- `Fetch`: A plugin for fetching the CVEs related to the Docker image.
- `Runner`: Individual scripts running as assertion techniques against the vulnerability in runtime.
- `Callback`: A notification means to the requestor.
- `Connection`: Underlying Host packages that enables the presentation OSI layer of the framework with a type of connection handled by the core engine, e.g., SSH.

Core is considered the core engine of the framework. It could be regarded as an orchestrator on demand to interface with all other components in the framework. Some of its features include orchestration, scanning engine, telemetry, analytics, scheduler, trust, and queuing. The emphasis in the component is given to the *scanning engine* where a substantial section of the core development takes place.

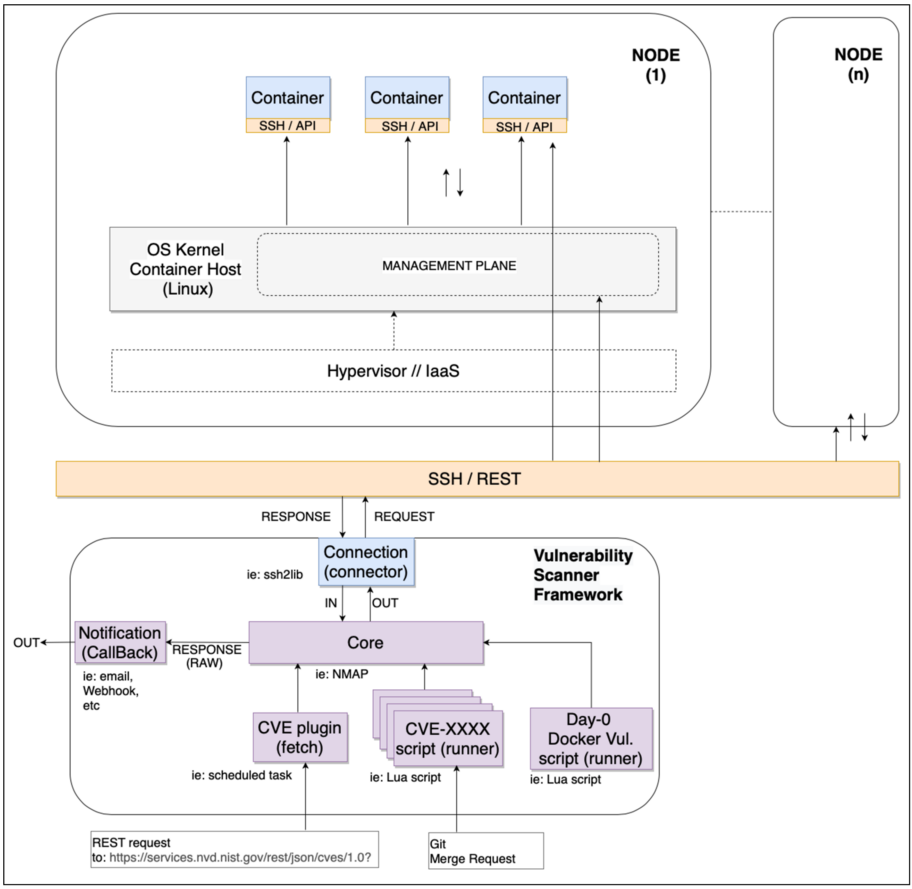

Fig. 1. Docker containers *vulnerability scanning framework*—Framework

VSF proposes to *black-list* containers that returned *true-positive* as a result of the assertion testing and also to those containers which has not been tested via VSF. On the other hand, if the assertion test results as a *true-negative* containers can be *grey-listed* and digitally signed; however, if the Accuracy Factor of the assertion testing is lower than 75%, then the image is *black-listed* due to its low certainty on the trust of the running container.

The container's digital certificate signing request will adopt the DevSecOps environment PKIs as required, and the certificate can be made available via the container's secret exchange method of choice.

The Accuracy Factor (AF) is a metric measured in percentage that indicates the certainty of the runtime assertion test completed given some conditions, including:

- An AF value lower than 75% results into a *black-listed* docker container;
- An AF value between 75% and 100% is considered as a true result;

– An assertion test that relates to a Docker container image other than the one described in the CVE.

Fetch, this component interacts with the NIST (National Institute of Standards and Technologies) APIs to retrieve relevant vulnerabilities classified as CVEs that may affect the Docker image. The key metrics to filter the rating includes (a) the exploitability score threshold, (b) the impact score threshold, and (c) keywords to match the Docker image. It permits a rapid manner parsing vulnerability information and/or detecting new exposures, across the Docker Host virtualisation layer and the application or Docker image. Docker images are pulled from the Docker Hub. The parsing criteria relate to the relevant Docker images are utilised in the environment. These can be denoted by a tuple (a, b, c), where ($a \geq 1.8$, $b \leq 5.9$, $c =$ "*docker*").

Figure 2 refers to results of the fetch module with 153 vulnerabilities identified given the values (a, b, c). In this example, the vulnerability identified as the number 81 relates to CVE-2016-3728.

```
*** ( 81 ) ***
CVE:  CVE-2016-3738
Description:  Red Hat OpenShift Enterprise 3.2 does not properly restrict access to
 STI builds, which allows remote authenticated users to access the Docker socket an
d gain privileges via vectors related to build-pod.
attackVector:  NETWORK
attackComplexity:  LOW
exploitabilityScore:  2.8
impactScore:  5.9
*********
Pages: 1, Total CVEs scanned: 153, Listed: 81 **
*********
***EOF***
```

Fig. 2. Fetch results—an example

The fetch module could be exchanged if the CVEs were unknown, with the DIVDS system proposed by Kwon et al. [8] amongst one of the passive models to detect impacting vulnerabilities in containers via *white-listing*. Hence, the *fetch* component is presented as a lightweight alternative that assumes pre-existing knowledge on the CVEs and exposures to Docker containers environment, typically the case in DevOps teams. However, VSF aims to *grey-list* despite the DIVDS approach.

Runner uses the *nmap scripting engine* (NSE) to execute the assertion testing on a specific vulnerability relevant to the Docker Host environment or Docker image. The Art defines the artifacts that include each vulnerability assertion test in an independent NSE script. The NSE scripts are executed in runtime within the Docker environment. The *Runner* component contains a collection of NSE scripts that will be classified based on the CVE id. Should the *Core* component consider necessary to validate this vulnerability, it would then execute the *nmap* script and obtain the response of the assertion test. Results are to be retrieved and identified as binaries, *true-positive* or *true-negative*.

An example of a *Runner* NSE script for CVE-2020-35195 is detailed below. It conducts assertion testing against the existence of a blank root password in the container in runtime for a *haproxy* image.

```
if conn:password_auth(user, passw) then
  local A1 ... = conn:run_remote(cmd)
  local A2 ... = conn:run_remote(cmd2)
  local _A2, _y = string.find(A2, _blankpw)
if _A2 == 1 then
  stdnse.verbose("Passed␣assertion␣test")
return "CVE␣assertion␣test␣Passed"
```

Callback relates to a notification mechanism to inform on the outcome of the assertion testing completed by the *runner*. The *core* component would have captured the assessment completed, and this is to be reported asynchronously back to the requestor. Notifications are to be sent as webhooks.

Connection contains underlying Host OS packages required to interact at a network layer with the Docker host, in order to enable connectivity of the *core, runner* components. Two options can be chosen, including *libssh2* and *openssl*.

The components of the *vulnerability scanning framework* (VSF), as previously defined, present a lightweight novel security framework for Docker container environments. Its purpose is to provide runtime Docker host and container's vulnerability software scanning capability to offer Privacy and Data Protection via completing active assertion testing techniques that would *greylist* running containers. It assumes pre-existing knowledge of the CVEs, as in the case of DevOps teams. The modularity of VSF resides on its capability to exchange some of the methods, i.e., the *fetch* and *core* components, could be interfaced with the DIVDS system defined by Kwon et al. [8], or with the certificate exchange proposed by Guo et al. [5] respectively.

4 Experiments

We present the experiments in two sections, the underlying scanning engine with the *initial experiment*, followed by the *case studies* and subsequent *analysis*.

4.1 The Scanning Engine Selection—Experiment

The *initial experiment* for the vulnerability scanner consisted of testing the underlying agent that would be running the assertion testing. In this case, we selected *tsunami* and *nmap* (NSE) as two well-known tools in the open-source community.

Our experiment with *tsunami* and *nmap* comprised of installing the each tool on disparate AWS EC2 CentOS 7.0 instances. The execution time for *tsunami* is comparable to those seeing in *nmap*, with 12.355 s to run. Table 2 compares the two scanning engines, with the capabilities as detailed below:

- *Open source*, *nmap* is a well-known network security scanning tool; and *tsunami*, is a Google initiative that allows development of network security scanning.
- *Programmatic development*, is the capability to developing vulnerability assertion tests over a programmatic approach. *nmap* is implemented in Lua and *tsunami* in Java.
- *Response time*, is the execution time of a one scanning job. Both tools are of comparable execution time as per our experiment; with 12.355 s on *tsunami* and 1.86 s on *nmap*. This is a metric that can vary, and it is dependant on the complexity of the scan as observed later in the case studies.
- *Lightweight*, is the ability to consume the tool across a wide range of environment in a simple manner. The experiment revealed that *tsunami* has a large set of dependencies when compared to *nmap*; which makes *nmap* considerably easier to deploy and portable across environments. We consider as a result of the experiment that *nmap* is a *lightweight* tool.
- *Network port scanning*, is the ability to run active network port scanning on a target system. Both *tsunami* and *nmap* cover this capability.
- *Relevance in the Industry*, is the presence and relevance of the tool in the industry. We define this capability as a measure of the risk to adopt the tool given its wide spread in the industry. We consider both tools *tsunami* and *nmap* offer low risk.

The previous capability analysis offers comparable characteristics for the selection of the scanning engine, between *tsunami* and *nmap*. However, the lightweight capability of *nmap*, its execution simplicity, and portability distinctively enable the proposed framework for rapid development on its scanning component. Thus *nmap* NSE is VSF's *scanning engine*.

Table 2. Scanning engine comparison

Tools	Open source	Programmatic	Response time	Lightweight	Port scanning	Relevance
nmap	Yes	Yes	Yes	Yes	Yes	Yes
tsunami	Yes	Yes	Yes	No	Yes	Yes

4.2 Case Studies

The presented *case studies* assume that VSF is used by a DevOps team across various scenarios, including Standalone and CI/CD pipelines. Hence, CVEs are known or checked via the *fetch* component of VSF. The container signing request and secrets management methods detailed in the *core* component are not represented and outside of the scope of the experiments in the *case studies*.

The experiments follow the same execution principle against different running Docker containers, Docker host, and emulation scenarios. Firstly, we develop individual *nmap* NSE scripts per CVE. These scripts are written in Lua-NSE and

reference each relevant CVE attack vector against a target running container. Our approach consists of the following steps per NSE script as detailed below and as shown in Fig. 3:

Step 1, to complete a Docker host or Docker container network port scanning against common network ports relevant to the communications;
Step 2, to authenticate into the Docker host; the NSE script to authenticate into the Docker host using the root credentials;
Step 3, to gain privilege access into Docker host;
Step 4, to identify the relevant Docker container id (as optional step);
Step 5, to run Docker remote commands from the host into the target Docker container, or to run Docker local commands in the Docker host;
Step 6, to navigate through the conditions as defined in the impacting CVE that makes the running target container vulnerable;
Step 7, to match the assertion criteria for the impacting CVE;
Step 8, to present the scan result as *true-positive* or *true-negative* with an accuracy factor; then repeat from *Step 1* to complete next scan.

Direct access via the container's network interface (CNI) is not used, as in specific container images, this capability is disabled. As detailed before, the *Anatomy* of VSF includes the artefacts that leverage *nmap*'s ability to scan ports and define network connectivity (i.e., *libssh2*) to reach the container; then define the rule sets and followed by actions and logic to determine whether the running environment is vulnerable or not with an accuracy factor.

The accuracy factor (AF) is a metric we use to measure the certainty of the scan; i.e., if the scan targets the specific Docker container image relevant to a CVE and the matching criteria are validated, the AF value is 100%. However, if the vulnerability scan is run against a different image than the one referenced in the CVE, we estimate that the AF is 50%, as the certainty is lower given the CVE conditions are not fully met.

The *Art* defines the artefacts in the *runner* component, which include the development for the following shortlisted CVEs for Lua-NSE development:

- CVE-2020-35467, relates to a "Docker docs" container image vulnerability,
- CVE-2020-35195, relates to a "haproxy" container image vulnerability,
- CVE-2020-15157, relates to a "containerd" vulnerability,
- CVE-2016-3697, relates to a Docker "runC" vulnerability,
- CVE-2021-21284, relates to a "libcontainerd" vulnerability.

The performance of each case study is checked against one metric. It is defined as the *velocity* during scanning, which is the number of scans in the environment against the execution time in seconds. Depending on volume, this will incur in greater overall execution time that will impact the case study. $\overline{A}$ would be relevant within the DevOps team context as it impacts the pipeline execution.

$$\frac{\Delta\overline{A}}{\Delta\overline{t}} = \lambda$$

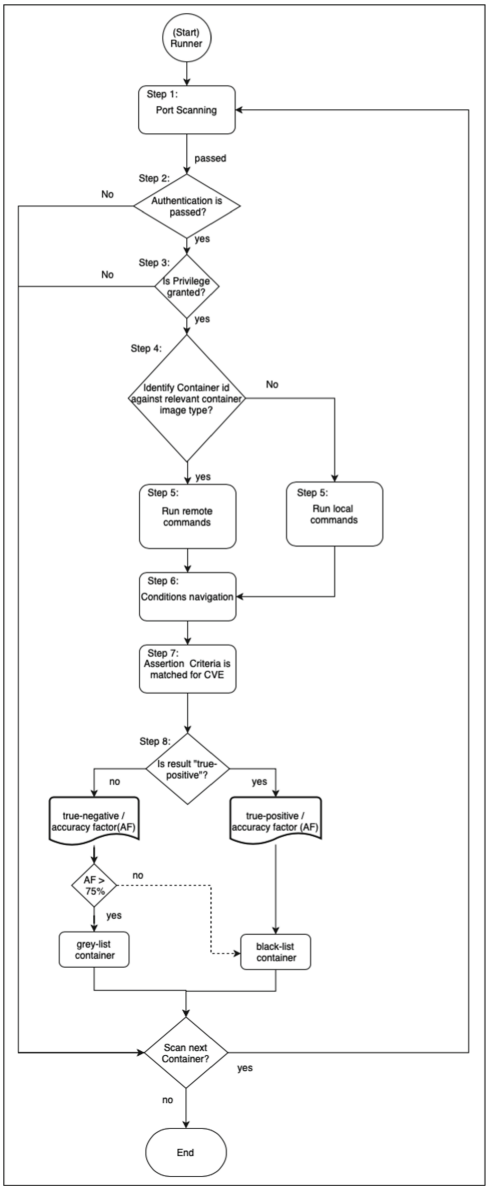

Fig. 3. Vulnerability scanning

– $\overline{A}$, the number of assertion tests in a period of time,
– $\bar{t}$, the execution time of a NSE script,
– λ, the number of test over period of time, velocity.

A *velocity* of 1.5 assertion tests is achieved via VSF given the detailed *runner* artefacts (Table 3). The velocity would vary depending on the vulnerability complexity and compute required in the execution of the NSE. The execution time does not fall into a distribution or statistical relationship as shown in Fig. 4; instead these are derived from the complexity of the vulnerability on the Docker ecosystem and its related attack vector complexity as execution time associated with the scan type.

Table 3. Experiment results: execution time

Scan type	Performance (s)
CVE-2020-35467	0.68
CVE-2020-35195	0.68
CVE-2020-15157	0.57
CVE-2016-3697	1.86
CVE-2021-21284	0.56

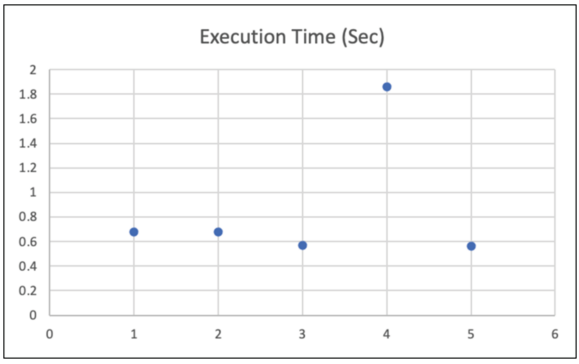

Fig. 4. Experiment results: execution time plot

The case studies are detailed as follows, where we act as a DevOps team across multiple scenarios.

Standalone, this case study encompasses the use of VSF in a standalone Docker environment, such as a software developer's IDE (integrated development environment). The case study assumes that an IDE runs in a Linux-based Operating System to leverage VSF components' capabilities. Our Docker container environment included the following running containers as active:

- *docker.io/centos*
- *docker.io/docs/docker.github.io*
- *docker.io/haproxy*

CVE-2020-15157, as demonstrated in Fig. 5 the NSE script shows a matching CVE condition which asserts a *true-positive* result in runtime with an AF value of 100%. Implying that the active Docker container is running a configuration in runtime that points to an external source, as per test case **A3** in Table 4. In this case instead of reading the offline image manifest, VSF validates the runtime foreign layer by reading the active *config.json* file per container on each *libcontainerd* process and returning a *true-positive* result, as per path: `/run/docker/libcontainerd/[id]/config.json`

Fig. 5. CVE-2020-15157—VSF scanning results

Similarly, we completed the CVE-2020-35195 and CVE-2020-35467 assertion testing procedures related to the *haproxy* and *Docker Docs* active containers as per test cases **A1** and **A2** respectively. Our assertion testing results (See Table 4) revealed that our running container are set with no-password for the root user, resulting in an AF value of 100% for each scan. Our VSF assertion demonstrates *true-positive* results, as per the path `/etc/shadow` and condition `root:!::0::::: .`

Our test case **A5** CVE-2021-21284, requires the utilisation of *namespaces* within the Docker containers environment via root user privilege access. In this test case, our standalone environment did not have enabled *namespaces* so resulted in nonvulnerable assertion testing as a *true-negative*. The recorded AF value is 80%, and this container would still be *grey-listed*. We refer to Table 4 as an impact on the Docker host.

```
 --userns-remap
# cat /etc/docker/daemon.json
{
 "userns-remap":"admin"
}
```

Finally, in test case **A4** related CVE-2016-3697 attempts to capture numeric UID as usernames. This test case resulted in non-vulnerable assertion testing as a *true-negative*, given that none of the containers captured these running conditions as per the below path. The recorded AF value is 100%.

`libcontainer/user/user.go`

Table 3 shows the maximum value to the run time execution time on the local CI/CD pipeline, which is 1.86 s. In addition, to identifying *true-positive* assertion test results against the CVEs rule sets as defined in VSF. Test Cases **A4** and **A5** relate to conditions within the Docker Host. Whereas **A1**, **A2** and **A3** relates to Docker containers as described in the experiments. The key benefits of the proposed methods arise in the ability to rapidly validate vulnerability exposures in the Docker Host and running Docker containers which as per results presented in Table 4.

Table 4. Experiment results: CVE assertion testing—*runner* component

Test case (runner)	A1: CVE-2020-35467
Test result	*true-positive*
Impact	container: docker.io/docs/docker.github.io
Performance	0.68 s
Accuracy	100%
Trust	*black-listed*
Test case (***runner***)	**A2**: CVE-2020-35195
Test result	*true-positive*
Impact	container: docker.io/haproxy
Performance	0.68 s
Accuracy	100%
Trust	*black-listed*
Test case (***runner***)	**A3**: CVE-2020-15157
Test result	*true-positive*
Impact	host: Docker host
Performance	0.57 s
Accuracy	100%
Trust	*black-listed*
Test case (***runner***)	**A4**: CVE-2016-3697
Test result	*true-negative*
Impact	host: Docker host
Performance	1.86 s
Accuracy	100%
Trust	*grey-listed*
Test case (*runner*)	**A5**: CVE-2021-21284
Test result	*true-negative*
Impact	host: Docker host
Performance	0.56 s
Accuracy	80%
Trust	*grey-listed*

As a summary of the experiment results, our standalone Docker environment is *black-listed* given the results of test cases **A1**, **A2** and **A3**. Also as per Fig. 6 the assertion testing accuracy is high with existing *true-positive* Docker containers. Hence, our IDE require container images updates or configuration updates to mitigate these risks as identified in the Docker environment.

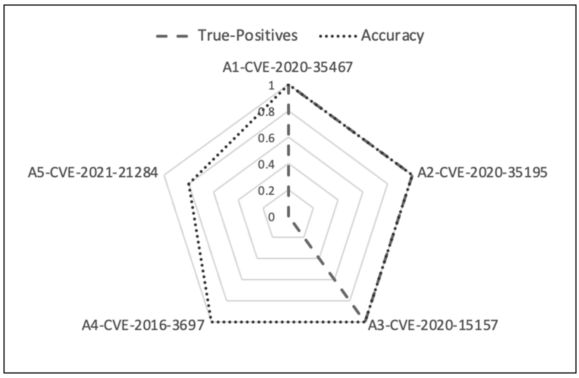

Fig. 6. Standalone assertion results VS accuracy

CI/CD Pipelines, this case study involves a dynamic insertion of VSF within a CI/CD pipeline tool-chain, with containers being used for infrastructure platform orchestration. The case study assumes we are a DevOps team that is using *Jenkins* as a typical CI/CD pipeline in a DevSecOps environment to manage infrastructure platform orchestration. Our Docker environment is used for code promotion during pipeline testing. The following container image was loaded in Docker as available in Docker Hub[1] (A 2 years old image).

In test case **B1**, we deploy the *Jenkins* Docker container and launch VSF in the Docker Host environment. VSF does not contain a specific CVEs NSE script for the *Jenkins* container image. Nevertheless, we complete runtime scanning against known vulnerabilities as we consider it relevant to this image. In our test case, we target the *Jenkins* workload as the target container and validate that the running container may have exposure as related to known vulnerability CVE-2020-25195 related to blank password condition for the root user, as per the path and matching condition shown below:

```
/etc/shadow
root:!::0:::::
```

Table 5 presents the result of the test case **B1**, and demonstrates the runtime execution of VSF. The assertion testing on the *jenkins* image against CVE-2020-35195 resulted in a *true-negative* and an AF of 80% due to the condition that the scan CVE is related to a different image *haproxy*. Our testing classifies the *jenkins* container as *grey-listed*.

[1] *docker.io/jenkins:2.60.3*.

Table 5. Experiment results: CVE assertion testing against *jenkins* image—*runner* component

Test case (*Runner*)	**B1**: CVE-2020-35195
Test result	*true-negative*
Impact	container: docker.io/jenkins:2.60.3
Performance	0.71 s
Accuracy	80%
Trust	*grey-listed*

5 Analysis

The case studies demonstrated how Docker active containers are *grey-listed* or *black-listed* against relevant CVEs. The accuracy factor (AF) is found as a required metric to validate the assertion test upon a criterion within VSF. It was proven to be an effective metric in test case C1 when a *grey-listed* container was rated with an AF of 80% equal to a lower certainty.

The experiments presented a velocity of $1.15 \frac{scans}{sec}$ and the samples captured as per the plot shown in Fig. 4 does not fall into a statistical relationship. However, the execution time is directly proportional to the complexity of the vulnerability on the Docker ecosystem and its related attack vector complexity as the CVE detailed in the *fetch* component as per Fig. 2.

The first case study was the **standalone**, where a typical IDE environment for a DevOps team member pulls containers from Docker Hub. VSF offered a capability to rapidly complete CVE assertion testing in runtime to validate if the running container is vulnerable. Table 4 present the result where *true-positive* events were detected on test cases **A1**, **A2** and **A3**. Test Case **A3** CVE-2020-15157 matches the CVE condition which asserts in runtime that the Docker image is running a configuration from an image that points to an external source in a known CVE. Consequently this container is *black-listed* until resolved and VSF scan presents a *true-positive* result with an AF value of 100%, as a defence mechanism to vulnerable container images.

The **CI/CD pipeline** applied test cases **B1**. In this case, the DevOps teams considered that the condition was relevant for *Jenkins* container images, with an AF factor of 80%. The trust result shows that the *Jenkins* container is *grey-listed*. It is indicative of the potential, portability, and flexibility of VSF for DevOps teams.

Table 6 presents a **comparative analysis** across four key capabilities, including *scanning, defense, attack* and *runtime*; when compared to related work as per previous sections. VSF demonstrates a robust reach across the security portfolio of a DevSecOps environment as the only tool with three categories, including scan, defense, and runtime. Even though the runtime capability overlaps with other frameworks, VSF is unique in its ability to complete Docker Host and container's **software** vulnerabilities scanning in runtime, not evidenced in related work.

Table 6. Comparative analysis

Related work/Capability	Scanning	Defence	Attack	Runtime-Hardware	Runtime-Software
VSF	Yes	Yes	No	No	Yes
Two-stage defense approach [3]	No	Yes	No	Yes	No
DIVDS [8]	Yes	No	No	No	No
Docker thread detection framework [6]	Yes	Yes	No	No	No
SecCDS [7]	No	Yes	No	No	No
Security assessment framework [11]	Yes	No	No	No	Yes
UBCIS [2]	Yes	No	No	No	No
Managed container framework [1]	No	Yes	No	No	No
Container state attestation [5]	No	Yes	No	Yes	No

6 Results

VSF's execution performance in runtime has no impact on the execution time of the container. In fact, the execution time and scanning velocity of the *vulnerability scanning framework* VSF are directly proportional to the complexity of the vulnerability and its attack surface.

The experiments consisted of two stages. In the first stage, the focus was to baseline the *scanning engine* tools that can effectively perform security vulnerabilities scans against container environments running Docker. The second stage consisted of utilizing the tool in a software-driven approach over a proposed framework (VSF) to *grey-list* or *black-list* active Docker containers as a trust model.

Nmap NSE was *scanning engine* shortlisted due to its lightweight capability on the initiation of the engine. Our testing revealed that *tsunami* could perform scanning functions; however, its interdependencies in the installation may encounter issues for adoption as a lightweight tool. Whereas *nmap* is widely available as a native tool in many security infrastructures, libraries are available and have proven their usability as per the case studies. VSF leverages *nmap* NSE for containers *privacy* and further covers the capability of detecting vulnerabilities that would affect the containers and its *data* from know exposures as detailed in CVEs.

Our case studies obtained a trust posture of each Docker environment presented via the assertion testing of the VSF engine that resulted in *grey-listing* or *black-listing* the target Docker container. The trust was cross-referenced against an accuracy factor that indicates the certainty of the result given explicit conditions. The combination of the two metrics allowed us to obtain a runtime vulnerability assessment of each Docker container. In addition, the flexibility of VSF has proven that it can be used on multiple scenarios, with a central view on the DevSecOps environment.

6.1 Recommendations

VSF has proven to fulfill a function not found in related work due to its software vulnerability scanning capability in runtime for Docker containers. Whether used

as a framework or as a discrete tool per component, it can complement other security initiatives within a DevSecOps strategy agenda as a lightweight tool for a Docker containers environment. Some highlighted recommendations include:

1. On the *Framework*, by improving the credentials management during assertion testing; and adding a containers' signing request capability as a functional addition to the framework.
2. On the *Usability*, by facilitating an easy deployment approach for DevOps teams to deploy the tool in DevSecOps environments; and by adding automation to VSF in order to facilitate the operational requirements of the adopting team.
3. On the *Accuracy Factor model*, by gathering a larger data set that allows for a higher volume of assertion testing results.

7 Conclusions

This paper proposed the *vulnerability scanning framework* (VSF) as a novel lightweight toolset for DevSecOps environments. VSF targets DevOps teams as the key audience. VSF delivers software vulnerability scanning capability to Docker container environments in runtime, leveraging *nmap* NSE scripts to deploy discrete assertion testing to relevant CVEs.

Further work is required on Docker containers security to enrich the trust model in runtime and orchestration. We will adapt VSF components to fit specific requirements generic to DevOps teams in software development CI/CD pipelines or infrastructure as code scenarios. Improving the trust model via the *greylisting* objective will also improve its accuracy factor module. Last but not least, we will add the feature of containers signing requests to the Docker containers environment.

References

1. Aguilera, X.M., Otero, C., Ridley, M., Elliott, D.: Managed containers: a framework for resilient containerized mission critical systems. In: Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD 2018), pp. 946–949 (2018)
2. Berkovich, S., Kam, J., Wurster, G.: Ubcis: ultimate benchmark for container image scanning. In: Proceedings of the 13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 2020), co-located with USENIX Security 2020 (2020)
3. Gao, X., Gu, Z., Kayaalp, M., Pendarakis, D., Wang, H.: Containerleaks: emerging security threats of information leakages in container clouds. In: Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2017), pp. 237–248 (2017)
4. Google: Tsunami. https://github.com/google/tsunami-security-scanner. Accessed 3 Mar 2021

5. Guo, Y., Yu, A., Gong, X., Zhao, L., Cai, L., Meng, D.: Building trust in container environment. In: Proceedings of the 2019 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE 2019), pp. 1–9 (2019)
6. Huang, D., Cui, H., Wen, S., Huang, C.: Security analysis and threats detection techniques on docker container. In: 2019 IEEE 5th International Conference on Computer Communication, ICCC 2019, pp. 1214–1220 (2019). https://doi.org/10.1109/ICCC47050.2019.9064441
7. Kong, T., Wang, L., Ma, D., Xu, Z., Yang, Q., Chen, K.: A secure container deployment strategy by genetic algorithm to defend against co-resident attacks in cloud computing. In: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 1825–1832 (2019). https://doi.org/10.1109/HPCC/SmartCity/DSS.2019.00251
8. Kwon, S., Lee, J.H.: DIVDS: Docker image vulnerability diagnostic system. IEEE Access **8**, 42666–42673 (2020)
9. Li, Z., Jin, H., Zou, D., Yuan, B.: Exploring new opportunities to defeat low-rate DDoS attack in container-based cloud environment. IEEE Trans. Parallel Distrib. Syst. **31**, 695–706 (2020). https://doi.org/10.1109/TPDS.2019.2942591
10. Martin, A., Raponi, S., Combe, T., Di Pietro, R.: Docker ecosystem-vulnerability analysis. Comput. Commun. **122**, 30–43 (2018)
11. Mostajeran, E., Mydin, M.N.M., Khalid, M.F., Ismail, B.I., Kandan, R., Hoe, O.H.: Quantitative risk assessment of container based cloud platform. In: Proceedings of the 2017 IEEE Conference on Application, Information and Network Security (AINS), pp. 19–24. IEEE (2017)
12. Schwarz, M., Lipp, M.: When good turns evil using intel SGX to stealthily steal bitcoins. In: Black Hat Asia 2018 (2018). https://i.blackhat.com/briefings/asia/2018/asia-18-Schwarz-When-Good-Turns-Evil-Using-Intel-SGX-To-Stealthily-Steal-Bitcoins-wp.pdf
13. Sultan, S., Ahmad, I., Dimitriou, T.: Container security: issues, challenges, and the road ahead. IEEE Access **7**, 52976–52996 (2019)
14. Xu, Q., Jin, C., Rasid, M.F.B.M., Veeravalli, B., Aung, K.M.M.: Blockchain-based decentralized content trust for docker images. Multimedia Tools Appl. **77**(14), 18223–18248 (2017). https://doi.org/10.1007/s11042-017-5224-6
15. Yasrab, R.: Mitigating docker security issues. arXiv preprint arXiv:1804.05039 (2018)