Concordia Institute of Information System Engineering (CIISE)
Concordia University

INSE 6130 OPERATING SYSTEM SECURITY

Project Progress Report:

IMPLEMENTING RECENT ATTACKS AND SECURITY ON CONTAINER

Submitted to:
Prof. Suryadipta Majumdar

Submitted by:

Student Name	Student ID
Anita Francis Archibong	27729790
Mustafa Talha Ucar	40059335
Eyiba Precious	40157231
Oladeinde Sukurat	40181568
Riya Vinodbhai Patel	40224858
Sanchit Smarak Behera	40230269
Hulli Rahul Ravi	40234542
Jubin Raj Nirmal	40235087

CONTENTS

PREFACE

The use of Docker containers has increased rapidly over time due to its flexibility to be deployed on multiple systems and be used as individual end points, but they are just as much vulnerable against end points attacks as of any other systems. End-Point Protection has been the need of the hour since a very long time in the Cyber World. Various defensive strategies have been devised to implement security features for the benefit of the client systems - this applies to docker containers as well. The present study is an attempt to testify basic attack strategies that the container systems face most commonly, analyze the degree of compromise that the attack achieves, and apply various known defensive strategies so that we can testify the number of defense strategies which are most effective against a particular attack.

INTRODUCTION

1 Overview

Docker[1] has become increasingly popular in the IT industry due to its ability to enhance software development and deployment. The project was initially developed in 2010 as an internal initiative at dotCloud, a PaaS company, with the aim of simplifying application packaging and deployment within dotCloud's PaaS environment. In 2013, Docker was released as an open-source project and quickly gained momentum among developers for its capacity to create self-contained, lightweight, and portable environments for running applications.

Over time, Docker has received significant adoption and backing from major technology companies, such as Google, Microsoft, and Red Hat. In 2019, Docker disclosed its decision to concentrate on its core technology and divest its enterprise business to Mirantis.

1.1 Docker Key Features

1. **Portability**: Docker containers can run on any platform that supports the Docker runtime, making it easy to deploy applications across different environments.
2. **Isolation**: Docker containers are isolated from the host system and from other containers, ensuring that applications run in a secure and controlled environment.
3. **Reproducibility**: Docker images are immutable and can be versioned, making it easy to reproduce and roll back changes to the application and its dependencies.
4. **Scalability**: Docker containers can be easily scaled up or down to meet changing demands, making it easy to manage resources efficiently.
5. **Flexibility**: Docker supports a wide range of programming languages, frameworks, and tools, making it easy to build and deploy a variety of applications.

1.2 Docker Benefits

1. **Simplified Deployment**: Docker makes it easy to deploy applications in a consistent and repeatable way, reducing the risk of errors and downtime.
2. **Reduced Resource Consumption**: Docker containers are lightweight and efficient, making it possible to run multiple containers on a single host system and reducing the need for dedicated servers.
3. **Improved Collaboration**: Docker images can be easily shared and reused, making it easier for developers to collaborate on projects and share best practices.
4. **Increased Productivity**: Docker's simplified deployment process and efficient resource usage can help teams deliver applications more quickly and with fewer errors.
5. **Version Control**: Docker provides version control for application containers, allowing developers to easily manage changes and rollbacks.

2 Concerns

Containerization[2] is a technology that has become increasingly popular in recent years, particularly with the rise of Docker and other containerization tools. While containers offer several benefits, such as portability, scalability, and efficiency, there are also concerns associated with their use. However, one of the primary concerns associated with containerization is security. Containers share the same host operating system kernel, which means that a vulnerability in one container can potentially compromise the entire system. It is therefore essential to properly configure and manage containers to ensure that they are secure from exploits.

3 Vulnerabilities

Below are listed some of the vulnerabilities in Docker[3][4] that can be exploited by attackers.

3.1 Insecure docker configuration

Docker configurations such as container networking, data volumes, and user permissions can be configured insecurely, leading to potential security breaches. This can happen by exposing unnecessary ports that attackers can use to gain entry, using weak authentication that can be easily exploited, communicating with insecure protocols that allow for interception and unauthorized access, granting excessive permissions that enable attackers to perform unauthorized actions, and running outdated software or operating systems with known vulnerabilities that can be exploited

3.2 Docker image

The process of creating containers often involves utilizing a base or parent image, which allows for the reuse of its components rather than building a new image from scratch. Docker images can contain vulnerable packages, libraries, and configurations that can be exploited by attackers to gain unauthorized access or cause harm to the system when an insecure image is employed.

3.3 Container breakouts

Container breakout vulnerabilities allow an attacker to break out of the Docker container and access the host system, which can result in significant damage to the system and sensitive data. When a malicious actor manages to bypass the isolation of a container and gain access to resources on the host system, it is known as container breakout. This can enable the actor to escalate their privileges and launch additional attacks once they have obtained root access on the host. To prevent such incidents, developers can utilize Docker security scanning tools to identify container breakout vulnerabilities, along with other types of vulnerabilities, and address them proactively before they can be exploited.

3.4 Free communication amongst containers

In a system with multiple containers running on the same host, facilitating communication amongst these containers is needed to achieve goals. However, this can become problematic as containers have a transient lifespan and implementing firewalls to prevent information leakage can be challenging. Therefore, the primary objective should be to reduce the attack surface by allowing only limited communication between containers.

3.5 Denial of Service

A Denial of Service (DoS) attack on Docker can occur in various ways, such as resource exhaustion, network flooding, and container-specific attacks. An attacker can exploit vulnerabilities in Docker containers or Docker hosts to cause a DoS attack, making the system unable to respond to legitimate requests.

3.6 Rogue containers

Containers that are running on the Docker host without proper authorization or permission. These containers are often created by attackers who exploit vulnerabilities in the Docker environment to gain unauthorized access to the host system. Rogue containers can pose a significant security risk to the Docker environment and the host system. They can be used to steal data, launch attacks against other systems, or disrupt the Docker environment.

3.7 Privileged containers

A privileged container is granted escalated privileges, allowing them to access and modify resources on the host that are normally restricted to other containers. By default, Docker containers are run in a

sandboxed environment with limited access to the host system, but privileged containers have unrestricted access to the host's resources, making them more powerful but also more dangerous.

ATTACK IMPLEMENTATION ON CONTAINERS

1 Software requirements

1. **Docker:** It is an open-source containerization platform. It allows users to bundle apps within containers. Docker helps execute new containers in very little time and is comparatively lightweight.
2. **Oracle VirtualBox**[5]**:** A hypervisor software that lets a user run different operating systems, at once.

2 Operating Systems used

1. **Kali Linux**[6]**:** The attack endpoint. Kali Linux has an advantage of having all major hacking tools, so comes in handy in most cases.
2. **Linux Mint XFCE**[7]**:** The user endpoint which has docker installed. The one benefit this operating system provides is that, apart from being very stable, it also is very light weight in terms of software configurations, this hinders any outside software end disturbances to the functioning of the docker environment.

3 Implementation details

Attacks: While working on the coursework we got to know some of the attacks on container as:

- **CVE 2019-5736** [8][11] - **RunC Attack:** Tampering (by malicious actor) of the runC binary on the docker host system with the help of a docker container running as a root, results in root access on the host system itself. Tampering here is done by replacing the runC binary, with a binary in the container, which is /bin/sh with the #! /proc/self/exe, which is a symbolic link to the runC binary on the host. While a malicious actor overwrites runC, he/she can inject malicious payload too, for example a backdoor/reverse_shell.
- **Abusing exposed Docker Registry APIs** [9]**:** Here we make use of exposed docker registries. Adversaries make use of exposed registry APIs to access private images stored inside the registry, which can be later used to compromise the systems that rely on this exposed registry to deploy containerized applications. In short, If you have misconfigured permissions in your docker socket file, or vulnerabilities in the docker daemon, the hacker can exploit it and perform this attack. The attacker gains access to the docker socket and uses it to gain privileged access to the host system.
- **CVE-2019-14271*** [10] - **Container escape using docker sockets:** The attacker gains root access to the host system and exploits the socket . The attacker then modifies system files and installs malware's and also gain administrative privileges.
- **CVE-2018-15664** [12] - **Privilege escalation using volume mounts/docker group:** When a volume is mounted, the contents of the volume become available to the host system, and the permissions on the volume are inherited by the mount point. If an attacker can gain access to a volume with elevated permissions, they can potentially leverage those permissions to escalate their own privileges on the system.
- **Denial of Service** [13] – The hacker perpetrates a network and makes a machine or resource unavailable for future or current use. This is a straightforward attack and, in most cases, the

resources which relate to computer network functions are targeted, so that the entire network of computers break down.

** There are two types of this attack, this type relates to exploit the socket and installs malware.*

*** This is the second type of this attack. Here, the attacker downloads docker images or credentials. They can also upload their own images to the victim's system.*

4 Successfully performed attacks

Sr. No.	CVE	Attack	By
1	CVE-2019-5736	RunC Attack	Jubin Nirmal
2	-	Container escape using docker sockets	Anita Francis
3	-	Privilege escalation using volume mounts	Anita Francis
4	-	Privilege escalation using the docker group	Oladeinde Sukurat
5	-	Abusing exposed docker registry	Riya Patel

4.1 RunC Attack

There are various approaches towards this attack. Here, we agreed on "attacker-controlled image" approach.

We have an ubuntu system (18.04.6 LTS) running docker (18.06.0-CE) with runC library 1.0-rc6. We git clone the repository mentioned above for the PoC.

The repository contains a malicious image folder which includes a script to overwrite the runC, a payload script for spawning a reverse shell at port 2345 and finally a Docker file which compiles everything inside the folder to create an attacker-controlled image, which when run on the host system results in gaining the root access.

4.2 Container escape using docker sockets

Suppose we are an attacker who has gained access to a container with the Docker UNIX socket mounted, and we have obtained a shell within that container. Our objective is to retrieve a file named "hackme.txt" that resides in the root directory of the host machine and can only be accessed by the root user.

1. Create a file named "hackme.txt" in the root directory of the host user and write "I am from host" into it using the nano command. Next, create a directory named "Dockersock" and navigate to it using the "cd" command to set it as our working directory.
2. We begin by initiating a container through the use of an alpine image and acquiring a shell, all while mounting the docker socket located at /var/run/docker.sock. To obtain the container ID, the docker ps command is utilized.
3. We need to verify if a Docker UNIX socket is properly mounted onto the container and proceed to install the Docker cli client within the container. Once installed, we can execute docker commands on the container. This will enable us to launch a new container on the host by utilizing the Docker socket and Docker client. Afterwards, we can attach the root directory of the host machine onto the newly launched container and gain access to the host's root directory by opening a shell within the container.
4. Within the given image, we designated the location of the Docker socket to be at /var/run/docker.sock and used the "-v" option to enable the mounting of the host's root directory beneath the container being initiated. The container was given the name "test". At the end of the command, we included "sh" as an argument to instantly acquire a shell within the container. We

navigated to the "test" folder where the host machine's root directory was mounted, and then changed our working directory to root. Finally, the ls command was used to display the contents of the root directory.

5. The image depicted below illustrates our successful breach into the file system of the host machine. We can confirm this by checking the contents of the "hackme.txt" file using the cat command. Thus, we have managed to gain access to a root-owned file on the host. This demonstration clearly exhibits the feasibility of leveraging the Docker UNIX socket, which is mounted on the container, to establish a foothold on the docker host.

4.3 Privilege escalation using volume mounts

1. We created a directory called "privilegeescalation" and accessed it. We attempted to view the contents of the "/etc/shadow" file, but the output indicated that we lacked the necessary privileges to do so.
2. To elevate the user's privileges, you need to create three separate files: Dockerfile.txt, shell.c, shellscript.sh. Next, you should add the following lines of code to each file, in the order specified.
3. After running the next command, the "shell.c" file will be compiled. Once the compilation process completes successfully, a new file called "shell" will be added to the list of files in the current directory. Following that, you can proceed to build a Docker image named "privilegeescalation".
4. Now the binary shell has been copied onto the image and the image has been built. The final line of the output reveals that the image has been labelled as "privelegeescalation:latest". With this accomplished, we can now initiate a container from this image and explore how we can utilize the groundwork we have laid to increase our privileges.
5. Once the container is launched, the earlier command will trigger the execution of the "shellscript.sh" file. This script is responsible for copying the "shell" binary into the shared directory and adjusting its file permissions. We can verify the contents of the "/tmp/shared" file on the host by running the subsequent command.
6. From the below illustration, we can observe that the file "/tmp/shared/shell" is possessed by the root user and has a "setUID" bit enabled. Hence, even if we execute this file with limited privileges, we should be able to run it with the same level of access as the root user.
7. Let us try to see the contents of /etc/passwd
8. We attempted to access the contents of /etc/shadow previously, but we were denied access due to our limited privileges. However, we ultimately were able to view /etc/shadow by virtue of a volume being mounted from the host into the container, whereby default processes run as root. To achieve this, we simply wrote a setUID root binary to the volume, which would then be recognized as a setUID root binary on the host.

4.4 Privilege escalation using the docker group

1. Create and add user ‘seun’ to the docker group.

   seun@ubuntu:`$ sudo usermod -aG docker seun
2. Exit to activate this new group, log back in and switch to user ‘seun’

   seun@ubuntu:~$ sudo usermod -aG docker seun
3. Open docker log in a new terminal

   seun@ubuntu:~$ su seun

   seun@ubuntu:~$ sudo journalctl -f
4. From the first terminal, pull docker image ubuntu with sudo, and image nginx without sudo. This is possible because ‘seun’ is a user in docker group and doesn’t need ‘sudo’ to run privileged commands.

   seun@ubuntu:~$ sudo docker pull ubuntu

   seun@ubuntu:~$ docker pull nginx

This shows that the command without sudo isn't logged, hence not traceable to the user 'seun'

5. Try to escalate the privileges available to user 'seun' by running a command as the root of the host machine from the docker container.

   seun@ubuntu:~$ docker run -it –name attacktest –privileged -v /:/host ubuntu chroot /host

With this, we get root access to the container which has also been mapped to the root of the host machine hence we can run commands to modify the host machine.

 # useradd 6130attack_test
#passwd 6130attack_test
Newpassword
Usermod -aG sudo 6130attack_test
Exit
seun@ubuntu:~$ id 6130attack_test

A new user was created on the host machine with sudo privileges without been logged, also this new user can log into the host machine with the set credentials.

4.5 Abusing exposed docker registry

While exploring vulnerabilities we got to know that, whenever the image is run from the docker registry the AIP of the registry gets expose. Base on that we tried to run this attack which is performed as follow:

1. First created the private Registry as "repository" and pushed few images
2. Checked which images, tags, image layers are present in the "repository" on browser by searching (Image was running on port 5000)

   localhost:5000/v2/_catalog
   localhost:5000/v2/<image_name>/tags/list
   localhost:5000/v2/<image_name>/manifest/<image_tag> (downloads text file in which will have sha26 hash representing the layer of the docker image)
   localhost:5000/v2/<image_name>/blobs/<sha256> (downloads a file, by inspecting the status of the docker image's filesystem in that specific layer can be discover)
3. After getting access the filesystem can be alter, here the image was corrupted.
4. The nginx image was runed as Target and "index.html" was removed and new file is added.
5. Then the modified image will create a new image with the command

   docker container commit
6. Lastly the new image will be pushed into the victim's repository, so when ever the victim tries to run the nginx image, the corrupted image will run.

**Refer Appendix for Screenshots*

DEFENSE IMPLEMENTATION AGAINST THE ATTACKS

1 Defense Overview

The defense team has been closely monitoring the attacks and trying to understand the anatomy of the attacks. However, attacks were recently implemented; therefore, the defence team needs more time to implement defense mechanisms.

There are several ways to detect vulnerabilities on containers:

- **Vulnerability Databases**: There are several vulnerability databases available online that contain information on known vulnerabilities. Please see the below table to see the CVEs that are used by the attack team.

No	CVE/ Name	Description	Defence Ideas
1	CVE-2019-5736	RunC through 1.0-rc6, as used in Docker before 18.09.2 and other products, allows attackers to overwrite the host runc binary (and consequently obtain host root access) by leveraging the ability to execute a command as root within one of these types of containers: (1) a new container with an attacker-controlled image, or (2) an existing container, to which the attacker previously had write access, that can be attached with docker exec. This occurs because of file-descriptor mishandling, related to /proc/self/exe.	The vulnerability affects the Docker versions before 18.09.2. Control root access to specific users only.
2	Privilege escalation using the docker group.	By default, a docker group is created during docker installation, any user added to the docker group will have elevated privileges to run the docker command without 'sudo' which can lead to privilege escalation. The docker log shows all the information and activities on a running container, this helps the administrator monitor or trace an event on the container. Without the sudo command, none of these commands will be logged. It is a problem if users are allowed to run docker command without being logged, because this vulnerability can be exploited to run malicious codes that can affect both the host and the docker container.	The vulnerability affects the Docker versions before 18.09.2. Control root access to specific users only.
3	Container escape using docker sockets.	A container escape using Docker sockets refers to a security vulnerability in Docker containers where an attacker gains access to the Docker socket and uses it to gain privileged access to the host system.	The vulnerability affects the Docker versions before 18.09.2. Control root access to specific users only.
4	Privilege escalation using volume mounts.	Privilege escalation using volume mounts involves exploiting the permissions associated with a mounted volume to gain elevated privileges on a system. When a volume is mounted, the contents of the volume become accessible to the host system, and the permissions on the volume are inherited by the mount point. If an attacker can gain access to a volume with elevated permissions, they can potentially leverage those permissions to escalate their own privileges on the system.	The vulnerability affects the Docker versions before 18.09.2. Control root access to specific users only.
5	Abusing exposed docker registry	If you have misconfigured permissions in your docker socket file, or vulnerabilities in the docker daemon, the hacker can exploit it and perform this attack. The attacker gains access to the	

		docker socket and uses it to gain privileged access to the host system.	

- **XDR & SIEM Tools:** Moreover, we were planning to use the Wazuh tool which can collect, normalize, and analyze security data from a variety of sources, including log files, network traffic, and system events. Wazuh supports containers as well. However, we realized that the system requirements for the Wazuh implementation are too high. Even in the documentation, it says: "We recommend configuring the Docker host with at least 6 GB of memory". Considering all our team members are using VirtualBox on our laptops, we decided not to use this tool.

2 Vulnerability Detection on Containers

Detecting vulnerabilities is a very important task. Vulnerable codes are how adversaries get unauthorized access to containerized application or to the host system. The detection of vulnerabilities on containers can be achieved through various methods, including manual inspection, vulnerability scanning, penetration testing, static code analysis, dynamic security analysis, and third-party audits. By utilizing these various methods, organizations can effectively detect vulnerabilities on containers and implement appropriate measures to mitigate risk and strengthen their overall security posture.

Manual inspection: Conducting manual inspection of container images, examining the software components and dependencies to identify known vulnerabilities. Manual inspection involves a thorough review of container images, with a focus on identifying known vulnerabilities within software components and dependencies. This could be by checking all accesses within the container manually, the ports open, the type of inflowing network packets, etc.

Vulnerability scanning: Employing vulnerability scanning tools such as Clair, Trivy, or Aqua, which can scan container images for known vulnerabilities. This would give us an overview on the technical flaws / loop-holes in the system, via which an attacker can take an opportunity to maliciously enter the system. What these vulnerability scanners do is compare the contents of the container image with a database of known vulnerabilities and report if any vulnerability is found. Monitoring new vulnerabilities is also very important. You can subscribe to vulnerability databases like the Common Vulnerabilities and Exposures (CVE), the National Vulnerability Database (NVD) to receive updates.

Penetration testing: Conducting penetration testing on containerized applications to identify potential security weaknesses and vulnerabilities. Penetration testing involves simulating an attack on a containerized application to identify potential security weaknesses and vulnerabilities. Pen testing should ideally be conducted with caution and in a controlled environment to avoid damage or disruption to the containers or their applications.

Static code analysis: Conducting static code analysis of container images, examining the source code for potential security flaws. Static code analysis examines the source code of container images for potential security flaws, while dynamic security analysis involves monitoring containerized applications during runtime for unusual or malicious behavior. Note that static code analysis is not a concrete method to detect vulnerabilities and should be used in a mix with other vulnerability detection methods, such as penetration testing and vulnerability scanning.

Dynamic security analysis: Running dynamic security analysis on containerized applications during runtime, monitoring for unusual or malicious behavior. This is a method of detecting vulnerabilities in containers by analyzing the behavior of the running application at runtime. Note that dynamic security analysis may impact the performance of the running containerized application. It should be run in a controlled manner (just like the penetration testing method), to avoid outages. It should be done regularly to mitigate any new vulnerabilities.

Third-party audits: Employing third-party auditing services to review and assess the security posture of containerized applications. third-party auditing services can be employed to provide an external review and assessment of the security posture of containerized applications. This can be an effective way to detect vulnerabilities in containers by bringing in an outside perspective and expertise. This method is costly and time-consuming and should be planned accordingly. It is important to select a reputable third-party to ensure that the auditing is done in the correct manner.

3 Enhancing the Defense of Containers

There are various strategies that can be employed to enhance container security. These include utilizing a minimal base image to limit the attack surface, ensuring that container images are kept up-to-date with the latest security patches, and enforcing the principle of least privilege by restricting container access to only essential resources. In addition, image scanning can be employed to identify vulnerabilities in container images prior to deployment. Secrets management can be implemented to securely store sensitive data, and runtime protection can be deployed to detect and prevent malicious activities in containers. Logging and auditing can also be implemented to monitor container activities and facilitate incident response. Through the implementation of these measures, the security of a container environment can be significantly enhanced. Let's probe into some measures that we can take to improve the security of the containers.

Use a minimal base image: Start your container with a minimal base image to reduce the attack surface. This means only including the necessary components and libraries required for your application to run. A minimal base image is essentially a stripped-down version of the operating system, and it includes only the necessary components to run the containers. Some of the major benefits include Reduced Attack Surface (Since there are no unnecessary services running on the system, the attack surface is reduced), Faster Deployment (Since the size of the base image is small, its easy to deploy), Improved Maintainability (there is a reduced risk of vulnerabilities), Improved Portability (This can be deployed easily to many different environments).

Update regularly: Keep your container images up-to-date with the latest security patches and updates. This can be achieved by regularly rebuilding and redeploying your containers. It is important to deploy the updates during the staging of the container before deploying it to other environments. This ensures that there will not be any issues with the containerized application. The tests you can do include both functional testing of the application and security testing, to ensure that the updates do not bring in new vulnerabilities. Staying up to dates with the latest threats is the best way to monitor and update your docker container. Additional benefits are to improve stability, adhere to compliance and staying up to day against latest threats.

Implement least privilege: Implement the principle of least privilege by limiting container access to only the necessary resources, such as network ports and filesystems. This can be achieved by using container orchestration platforms like Kubernetes, which provides features like pod security policies and network policies.

Enable Image Scanning: Enable image scanning to detect vulnerabilities in your container images before they are deployed. This can be achieved by using container image scanning tools like Clair, Trivy, or Aqua.

Implement Secrets Management: Implement secrets management to securely store and manage sensitive data, such as API keys, passwords, and certificates, used by your application. This can be achieved by using tools like Hashicorp Vault or Kubernetes secrets.

Implement Runtime Protection: Implement runtime protection to detect and prevent malicious activities in your containers. This can be achieved by using container security platforms like Aqua or Sysdig, which provide runtime protection and visibility into container activities.

Enable logging and auditing: Enable logging and auditing to monitor and track container activities. This can help detect and respond to security incidents in a timely manner. Tools like Fluentd or Elasticsearch can be used for centralized logging and auditing.

4 CVE 2019-5736

4.1 Understanding the skeleton of CVE 2019-5736

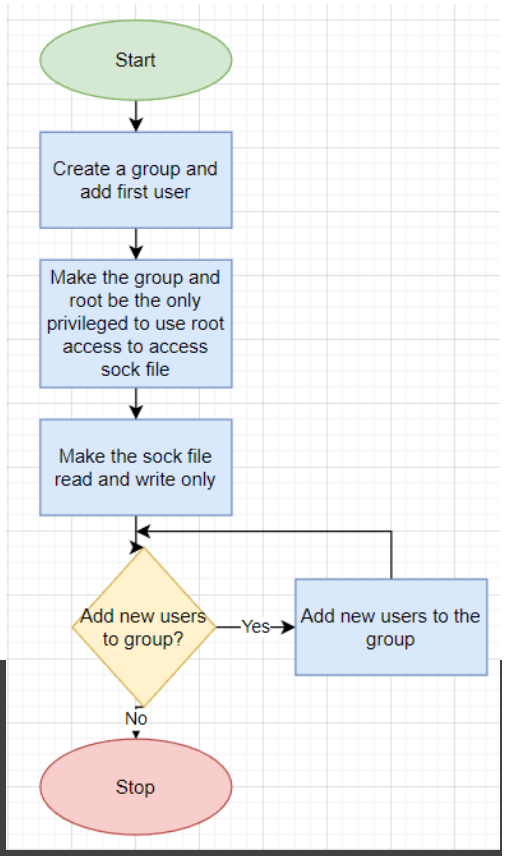
RunC attack is based entirely on replacing a runC binary with a malicious binary to gain root access of the container. (/bin/sh with the #! /proc/self/exe) [https://aws.amazon.com/blogs/compute/anatomy-of-cve-2019-5736-a-runc-container-escape/]
Note that this attack depends on a race condition. A race condition is a method of exploiting a vulnerability of a system with respect to the execution time. Suppose process A is modifying a file. Process B can modify this file before process A is done modifying. The race condition here is played between the two files: ***/bin/sh*** and ***#!/proc/self/exe***. Essentially the replacement of /bin/sh and #!/proc/self/exe is done here. The replacement gets halted by the system as this is not an allowed process, it needs root access.
Now, what does /proc do? **Proc** is a pseudo-filesystem that provides access to a number of Linux kernel data structures. Every process in Linux has a directory available for it which is designated with a **pid**. This directory shares a lot of information about the processes, which includes the arguments it was given when the program started, the environment variables visible to it, and the open file descriptors.
Note that /bin/sh is a file which accesses certain libraries. Libraries are made available to programs through linking. On Linux, programs can be statically linked, (linking done at compile time) or dynamically linked (linking done during runtime).
Where then is the vulnerability? /proc filesystem exposes a path to the original program's file even if that file is not located in the current **mount namespace**. Linux primitives such as namespaces typically requires you to run as root. In most installations involving runc, the whole setup runs as root. **RunC** must be able to perform a number of operations that require elevated privileges, even if your container is limited to a much smaller level of privileges.
So, as we know, /proc exposes a path to the original program's file and the process that starts the container runs as root. What if you have a program that is so important that you know it will run RunC? This is where the exploit happens. The race condition exploit will have one program which will try to update the runc file, and another will try to replace it with /proc/self/exe. There will be points where the writing of the file will render busy, but over repeated actions, the file will be replaced, and the adversary now gets complete control over the system as root.
Now, this begs the question: Is it this easy to run this exploit? Do containers not have protection against this attack? Containers are essentially known to isolate the host from the workload or workload from a host (**containerization**). Every container has a separate file system view, with a separate view, the container should not be able to access the host's files and should be able to see its own. Note that runc accomplishes this using a mount namespace and mounting container image's root filesystem as /. This effectively hides the host's filesystem.
Unfortunately, even after all of this, things do pass through the mount namespace. Additionally, the major issue is with the /proc filesystem itself.
In the view of the defender, ignoring all common fixes to this attack, the one thing that strikes the mind is: what if we know all our root users? Can there been only a set of users that can have a specific set of accesses? The design principle we are talking about is called **Separation of Privileges**.
Separation of Privileges ensures that your system runs in a controlled manner by controlling the kind of accesses the users have. We achieve this by categorization, and this fix requires a **bottom-up approach**, as discussed in the next section.

4.2 Defense Implementations for CVE 2019-5736

As we noted above the entire play starts when the adversary gains root access. The race condition is something that, no matter how hard you fight against, it can still be tackled with – long story short, it's simply a bad code idea. The fix we proposed here is to make sure we give only a specific set of users the root privileges and lock it to just them, in short, we control the privileges of what the users once they gain access. The more we harden the controls, the more we approach the design idea known as "Separation of Privileges". Make multiple divisions, each division has control over only a certain part of the container or process. The processes could be snowballing processes belonging to a major process or individual processes.

Now, let's go a little more further. If you make divisions for every single process on the container, this ensures that any new user that uses the container by default has zero privileges. Hence, we approach a design principle known as **fail-safe defaults** [].

The algorithm here would be as follows:



1: Start

2: Create a group and add first user: *Here we make a group (say docker_group) via the terminal and add our first user to it. This is necessary as we need that user to be able to add more users.*

3: Make the group and root be the only privileged to use root access to access sock file: *Sock file here is how the command line and sockets communicate. Here we make sure that the sock file is modified to be only accessible by any users in the group or root.*

4: Make the sock file read and write only: *The sock file is modified to 660, which means that the file is read and write only, not executable. Only the designated user can run this as executable.*

5: If add users to the group: Yes -> Add users to the group: *As stated, this is self-explanatory.*

6: If no -> Stop

The flowchart for the same is also provided.

Note that this fix is bottom-up approach. This means that we are fixing the vulnerability while sitting inside the container to the walls of the container. The opposite of this would be the top-down approach, where we sit outside the container and apply patches to the walls of the container with the variables outside the scope of that container

5 Privilege Escalatios

5.1 Understanding the skeleton of Privilege Escalations

Privilege escalation has the same goal as the RunC race condition attack – gain remote access. The way it does this though is by escalating the existing privileges on a specific user. There are multiple ways of doing this, and we chose three of them.

1. **Privilege escalation using volume mount:** The explanation to this is provided above. The steps of the attack are as below:

- A low privileged account is the target here, and the attacker gains access to that.
- Now, the attacker checks if there is any vulnerable service on the system that requires elevated accesses.
- The attacker now makes a malicious file and stores it in a volume that is controlled by it.
- The attacker mounts this volume on the target container.
- The attacker runs a command or process on the target system that executes the malicious file or payload from the mounted volume.
- The malicious code gains elevated privileges or system access, allowing the attacker to perform actions that were previously restricted, such as modifying system files or executing arbitrary code.

How do we fix this? Simple ensure that you don't allow unknown volumes to be mounted! But there is more to this than it seems. Let's address the other two cases.

2. **Privileged escalation using docker sockets:** The explanation to this as well is provided above, the steps are as below:
   - A low privileged account is the target here, and the attacker gains access to that.
   - The attacker checks if a Docker socket is mounted inside the container and has required permissions.
   - The attacker runs Docker daemon running on the host system to interact with that socket, thereby gaining elevated privileges.
   - The attacker may then use the elevated privileges to execute arbitrary code, access sensitive data, or compromise the host system.

Here the fix would be to enable TLSv1.3. TLSv1.3 analyses and recognizes traffic of such sorts; you can set an action against it then.

3. **Privilege escalation using the docker group:** Once again, explanation as above, steps as below:
   - A low privileged account on a Linux System is the target here, and the attacker gains access to that.
   - The attacker identifies if the user is a member of the Docker group, which is used to manage Docker containers.
   - The attacker uses the Docker group membership to execute Docker commands with elevated privileges, such as running privileged containers or mounting host directories inside the container.
   - The attacker may then use the elevated privileges to execute arbitrary code, access sensitive data, or compromise the host system.

Here, the fix is the same as the fix for RunC attack, make separate docker groups, add users and set privileges. The adversary will not be able to then use the docker group membership as now it has been halted by the Fail-Safe Defaults system design.

Note how all three are privilege escalations and all of them have different fixes since they are different methods. One of the fixes here fits for all the three, and it's the last stated fix. However, there is one more fix that should happen, and that is fixing the vulnerability of any code that may require elevated privileges, and in most cases, it's the docker cp command.

Docker cp command somehow has a vulnerability to execute a process which requires an elevated access, without really having to enter the password for it. Say if your command is

docker cp <<source>> <<destination>> | sudo su

You would directly get sudo access, no matter how trivial the files in cp are.

The fix to this would be to simply run docker cp in controlled environment, and one way to do that is to close the container, do the copy, open the container.

5.2 Defense Implementations for Privilege Escalations

1. We have three fixes here and let's focus on all of them. Create a separate docker group, add users to it, set privileges: Its exactly the same as the one for RunC, that will work for this issue.
2. Enable TLSv1.3: Simply update the configuration file of docker with TLSv1.3. This will be discussed in the fix for the next attack in question.
3. Make sure no unknown volume is mounted: By default, volume mounts should not be recognized. Any volume mount should be authenticated by the user.
4. Controlled use of docker cp: Below is how that algorithm looks like

```
Start
    Enter source path and destination path
    Do we have a source path?
        Store answer in A
    Do we have a destination path?
        Store answer in B
    While the copy is done
        If A is true and B is false
            Close container A
            Perform docker cp
            Open container A
        If A is false and B is true
            Close container B
            Perform docker cp
            Open container B
        If A and B are true
            Close container A and B
            Perform docker cp
            Open container A and B
End
```

6 Abusing Docker Registry

6.1 Understanding the skeleton of Abusing Docker Registry

This attack makes use of docker registries, as the name suggests. If you have misconfigured permissions in your docker socket file, or vulnerabilities in the docker daemon, the hacker can exploit it and perform this attack. The attacker gains access to the docker socket and uses it to gain privileged access to the host system.

The steps look as below:

- The attacker identifies a Docker registry that is accessible to the public or has weak authentication mechanisms. Weak authentication here would mean an easily guessable password, no two-factor authentication of any kind.
- The attacker uses tools such as docker pull or docker login to interact with the registry and attempt to gain access to private Docker images. The weak authentication is a straight give away for this attack here on.
- The attacker may use brute-force attacks, password guessing, or other methods to obtain valid credentials for the registry. This will not be tough for the adversary if the password in question can be broken via a mere dictionary attack.

- The attacker may then download and analyze the Docker images to identify any sensitive information, such as passwords or keys, that can be used to further attack the system. This is pretty straightforward.
- The attacker may also modify the Docker images to include malicious code, backdoors, or other forms of malware, which can then be propagated to other systems that use the same Docker images. This can cause a sort of mass attack which can render the entire company under this sort of attack a major loss in terms of time, and resources.

The fix here is very straightforward, “use strong authentication mechanism”. You would need to enter a non-guessable password, keep changing the password every 30-60 days, and enable two-factor authentication for logon. Docker services such as MySQL and PostgreSQL ensure this can be done, these services save passwords and are a safe bet.
But let us look at another fix we need to keep in mind: Enabling TLSv1.3. One major benefit of using TLSv1.3 is that it has the ability to recognize traffic and provide required outputs that can help mitigate any kind of issue that looks suspicious.

6.2 Defense Implementations for Abusing Docker Registry

Enabling TLSv1.3 on a docker system is a simple 3-fold process. There is a no real need for an algorithm for this, but this does have a flow.

1. Update the nginx file: Here we replace TLSv1.2 with TLSv1.3 in the nginx configuration file. The configuration file is how we tune the configurations of the nginx server.
2. Build the docker image: Post the configuration update, we would build the docker image, this makes sure that the changes in the configuration file are applied to the docker image.
3. Run the docker container mapping port 80 and port 443 to the container’s port: These two ports are used by TLSv1.3 and will be easier for it to understand the incoming traffic and filter it out.

Authentication mechanisms: This, again, is a pretty straightforward process. You need a service to store your passwords? MySQL and PostgreSQL do your work. These two services save passwords and ensure that these passwords do not leak. You also need a 2FA login to your containers. For this:

1. Go to your docker hub account
2. Go to Account Settings
3. Select Security tab
4. Select “Enable two-factor authentication
5. Select Set up using an app
6. Save your recovery code and store it somewhere safe

These two fixes more or less ensure the safety of your docker image and server against abuse of docker registry.

7 Patch Code

7.1 Understanding the algorithm of the patch code

The algorithms of each patch code are stated above. The question here is, should this be one major patch file or a set of small patch files? Ensure that when you write the codes, they are independent of each other in terms of utilization of variables to utilization of services, regardless these would be merged or not.
That being said, the major advantage of having the codes merged are as below:

1. There will be a one stop fix against all the attacks.
2. You do not need to worry about figuring out which code works for which attack.
3. One menu would be sufficient to be able to orchestrate the fix.

This also has a disadvantage:

1: One stop fix also means you might accidentally end up applying a fix which may not be on the affected area, which may introduce a new vulnerability.
Let's assume our users are lame, they could just run the same patch code, go through all the menu to figure out the fix blindly. This can lead to a major disadvantage as stated above. Hence, we need to make all the patch files individual against the attack.
Every patch file should then have a well-constructed documentation, so that

1. Our users can understand how the patch works
2. The developers of the future know how the patch file was made, so that it will be easy for them to figure out if any changes are necessary.

So, here, we will make the patch files individually against the attacks and operate them individually.
The patch files are made using python programming language, since:

1. Python is easy to code in and deploy: It's a very understandable language and easy to code and deploy.
2. Python libraries are powerful: A number of different functions are available, which makes the libraries intensive.
3. Python programs are quick fix and immune to race condition attacks: Self-explanatory.

7.2 Writing the patch code

7.2.1 RunC attack

Below is the code that works against RunC attack. We have made two patch files. One initiates the group creation; one adds new users to the group.

```
#!usr/bin/env python
import subprocess
user = input("You need to add the first user, name your first user")

subprocess.call("sudo groupadd docker-users")
subprocess.call("sudo usermod -aG docker-users "+user)

subprocess.call("sudo chown root:docker-users /var/run/docker.sock")
subprocess.call("sudo chmod 660 /var/run/docker.sock")

subprocess.call("sudo systemctl restart docker")
```

The line
sudo groupadd docker-users : Creates a new group
"sudo usermod -aG docker-users "+user : Adds the first user
sudo chown root:docker-users /var/run/docker.sock : Changes ownership of the sock file to the users in group and root
sudo chmod 660 /var/run/docker.sock : Modifies permission of sock file to read and write.

```
#!usr/bin/env python
import subprocess

user = input("Please state the name of the new user")
subprocess.call("sudo usermod -aG docker-users "+user)
```

The line

subprocess.call("sudo usermod -aG docker-users "+user) : Adds new users to the group

7.2.2 Controlled docker cp code

```
#!usr/bin/env python
import subprocess

sourcepath = input("Please share the source path")
destinationpath = input("Please share the destination path")
checkA = input("Is there a source container? Press 1 for yes, 0 for no")
if checkA == 0
   sourceContainer = input("Please input source container")
checkB = input("Is there a destination container? Press 1 for yes, 0 for no")
if checkB == 0
   destinationContainer = input("Please input destination container")

if checkA == 0 && checkB == 0
  break

if checkA == 1 && checkB==0
   subprocess.call("docker stop "+sourceContainer)
   subprocess.call("docker cp "+sourceContainer+":"+sourcepath+" "+destinationpath)
   subprocess.call("docker start "+sourceContainer)

if checkA == 0 && checkB == 1
   subprocess.call("docker stop "+destinationContainer)
   subprocess.call("docker cp "+sourcepath+" "+destinationContainer+":"+destinationpath)
   subprocess.call("docker start "+destinationContainer)

if checkA == 1 && checkB == 1
   subprocess.call("docker stop "+sourceContainer)
   subprocess.call("docker stop "+destinationContainer)
   subprocess.call("docker cp " +sourceContainer+ ":" +sourcepath+ " " +destinationContainer+
":" +destinationpath)
   subprocess.call("docker start "+sourceContainer)
   subprocess.call("docker start "+destinationContainer)
```

This is a pretty self-explanatory code; it exactly follows its algorithm stated above.

7.2.3 Enabling TLSv1.3

```
#!usr/bin/env python
import subprocess

#Update your docker file - replaces TLS 1.2 with TLS 1.3 in conf file
subprocess.call("RUN sed -i 's/TLSv1.2/TLSv1.3/g' /etc/nginx/nginx.conf")

#Build Docker Image
subprocess.call("docker build -t myimage")

#Run the docker container mapping port 80 and port 443 to the container's ports
```

```
subprocess.call("docker run -d -p 80:80 -p 443:443 myimage")
```

Explanation as provided in the code.

7.3 Implementation of the patch code against the CVEs

To apply the patch codes, we need to install python 3 on the containers. This can be done through a terminal command on linux.
sudo apt-get install python3
Or the install file can be downloaded via the python official page[].
These patch codes were applied to the docker files. We see two major observations:

1. The privilege escalations were denied as the “user needed elevated privileges”
2. This made changes to the required docker files and container files.

In short, we can conclude that these codes successfully worked against the attacks as intended.
Note that each attack was done under a specific container environment and the codes were deployed accordingly.

8 Conclusion

In conclusion, Docker has become an essential tool for software development and deployment in the IT industry. Its key features, including portability, isolation, reproducibility, scalability, and flexibility, have made it popular among developers and significant technology companies. However, container security has become a growing concern with the increasing adoption of containerization technology.
To test the container security, Various attacks were implemented, such as RunC attack that could allow attackers to escape the container and gain access to the host system, Privilege Escalation using docker sockets where An attacker who gains access to the Docker socket can potentially use it to launch containers with escalated privileges and execute arbitrary code on the host system, Privilege Escalation using Volume Mounts where An attacker who gains access to a container with a mounted volume can potentially access and modify files on the host system that are outside the container's scope, Privilege Escalation using docker groups an attacker gains access to a user account that is a member of the "docker" group, they may be able to use Docker commands to run containers with elevated privileges and abusing Exposed docker registry where an attacker could potentially gain access to any private images stored on the registry, including application code, credentials, and configuration files.
Various strategies were employed to enhance the security of containers, such as using a minimal base image, keeping container images up-to-date with security patches, enforcing the principle of least privilege, enabling image scanning to identify vulnerabilities, implementing secrets management to securely store sensitive data, implementing runtime protection to detect and prevent malicious activities, and enabling logging and auditing to monitor container activities.

Overall, by implementing these measures, the security of a container environment is significantly enhanced. However, it is essential to note that container security is a constantly evolving field, and as new threats emerge, new measures must be taken to mitigate them. Therefore, it is crucial to keep up with the latest developments in container security and continuously improve the security posture of container environments.
Docker has revolutionized the way applications are developed and deployed. As containerization continues to grow in popularity, we must prioritize container security and take proactive measures to ensure the security of our container environments.

Members Contribution

Our group of 8 members was divided into two teams: Attack Team with four members (Anita, Riya, Jubin, Oladeinde) and Defense Team with four members (Mustafa, Precious, Sanchit, Rahul)

<table>
<tr><th colspan="2">Attack Team</th><th colspan="2">Defense Team</th></tr>
<tr><th>Contribution</th><th>Member</th><th>Contribution</th><th>Member</th></tr>
<tr><td>RunC</td><td>Jubin Raj Nirmal</td><td>CVE 2019-5736</td><td>Mustafa Talha Ucar</td></tr>
<tr><td>Container escape using docker sockets</td><td>Anita Francis Archibong</td><td rowspan="2">Privilege Escalation</td><td rowspan="2">Eyiba Precious
&
Sanchit Smarak Behera</td></tr>
<tr><td>Privilege escalation using volume mounts</td><td>Anita Francis Archibong</td></tr>
<tr><td>Privilege escalation using the docker group</td><td>Oladeinde Sukurat</td><td>Abusing Exposed Docker Registry</td><td>Sanchit Smarak Behera</td></tr>
<tr><td>Abusing exposed docker registry</td><td>Riya Vinodbhai Patel</td><td>Code</td><td>Hulli Rahul Ravi</td></tr>
</table>

BIBLIOGRAPHY

[1] Docker Website: https://www.docker.com/
[2] Use containers to Build, Share and Run your applications: https://www.docker.com/resources/what-container/
[3] Vulnerability scanning for Docker local images: https://docs.docker.com/engine/scan/
[4] Docker vulnerability list: https://www.cvedetails.com/vulnerability-list/vendor_id-13534/product_id-28125/Docker-Docker.html
[5] Oracle VirtualBox: https://www.virtualbox.org/
[6] Kali Linux: https://www.kali.org/get-kali/
[7] Linux Mint: https://linuxmint.com/download.php
[8] RunC Attack CVE: https://nvd.nist.gov/vuln/detail/CVE-2019-5736
[9] Copy-On-Write Attack: https://nvd.nist.gov/vuln/detail/cve-2016-5195
[10] Container escape using docker sockets: https://nvd.nist.gov/vuln/detail/CVE-2019-14271
[11] RunC Proof of Concept: https://github.com/twistlock/RunC-CVE-2019-5736
[12] Privilege escalation using volume mounts: https://nvd.nist.gov/vuln/detail/cve-2018-15664
[13] Denial of Service: https://www.ncsc.gov.uk/collection/denial-service-dos-guidance-collection
[14] Abusing exposed docker registry: https://dreamlab.net/en/blog/post/abusing-exposed-docker-registry-apis/

APPENDIX

RunC

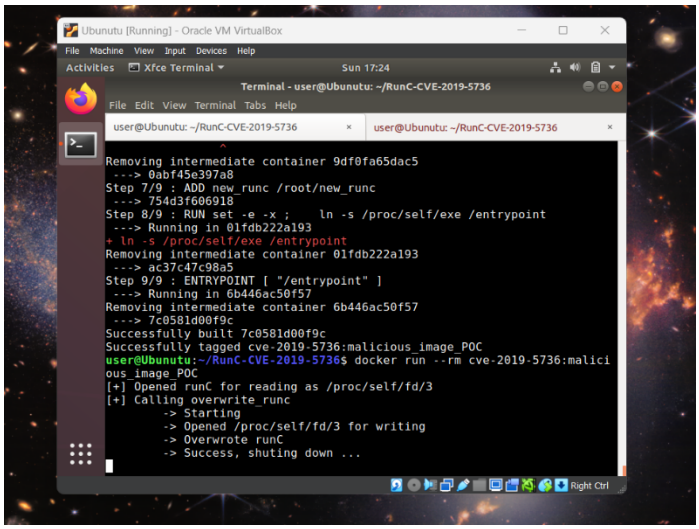


Fig 1: 4.1 RunC: Running the Attack

Container escape using docker sockets

Fig 2: 4.2 (1)

Fig 3: 4.2 (2)

Fig 4: 4.2 (3)

```
┌──(kali㉿kali)-[~/Dockersock]
└─$ docker exec -it 06e5e2993f73 sh
/ # ls
bin   dev   etc   home  lib   media mnt   opt   proc  root  run   sbin  srv   sys   tmp   usr   var
/ # ls /var/run/docker.sock
/var/run/docker.sock
/ # ls
bin   dev   etc   home  lib   media mnt   opt   proc  root  run   sbin  srv   sys   tmp   usr   var
/ # docker -H unix:///var/run/docker.sock run -it -v /:/test:ro -t alpine sh
/ #
/ # cd test
/test # cd root
/test/root # ls
hackme.txt
/test/root # cat hackme.txt
I am from host
/test/root #
```

Fig 5: 4.2 (4)

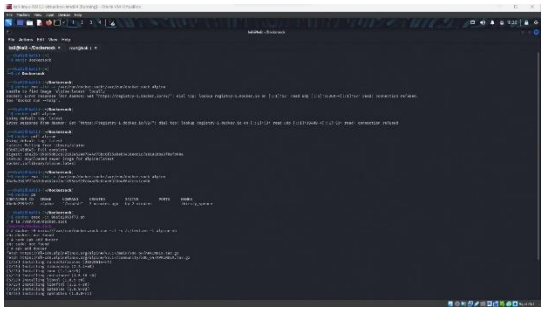


Fig 6: 4.2 (5)

Privilege escalation using volume mounts

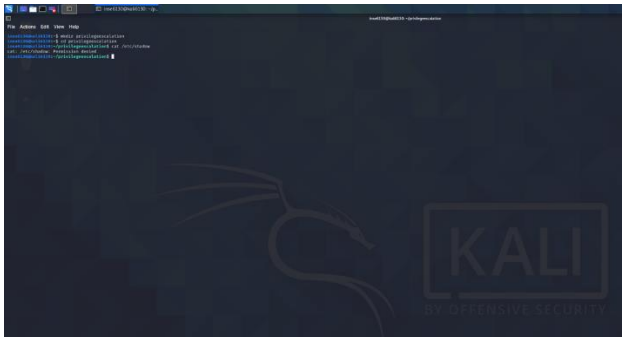


Fig 7: 4.3 (1)

Fig 8: 4.3 (2)

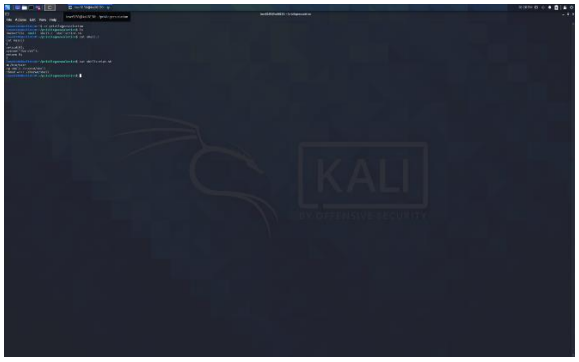


Fig 9: 4.3 (3)

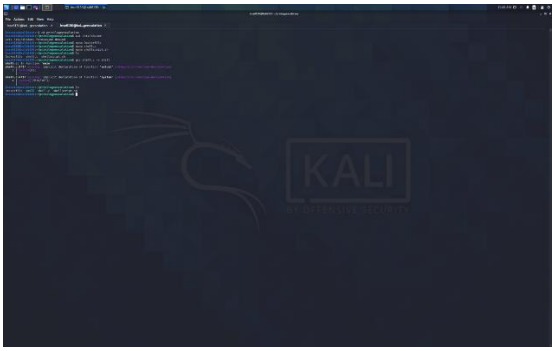


Fig 10: 4.3 (4)

Fig 11: 4.3 (5)

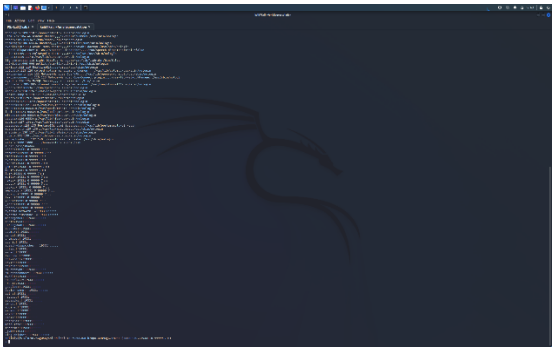


Fig 12: 4.3 (6)

Privilege escalation using the docker group

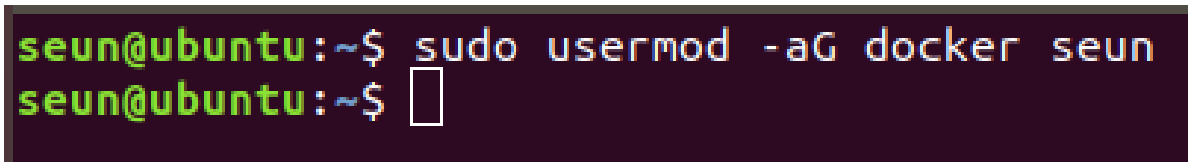


Fig 13: 4.4 (1)

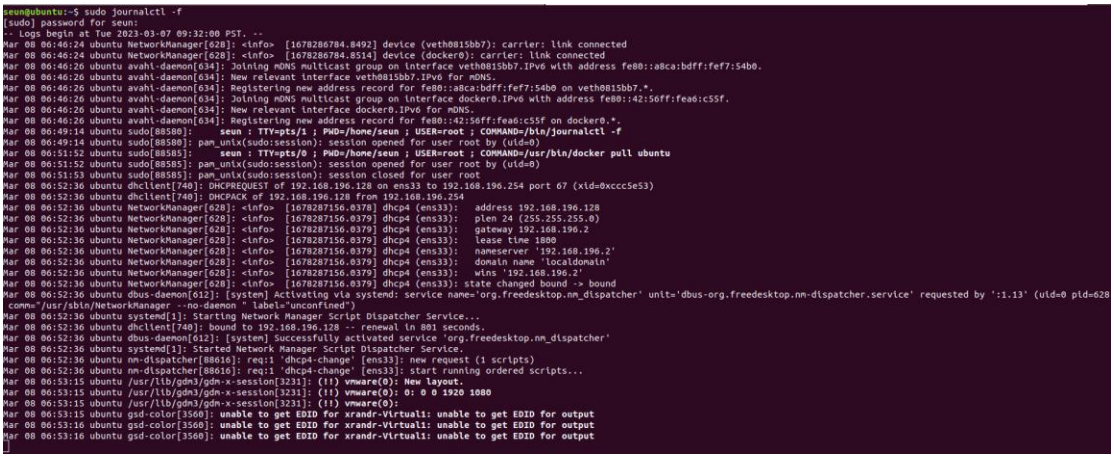


Fig 14: 4.4 (2)

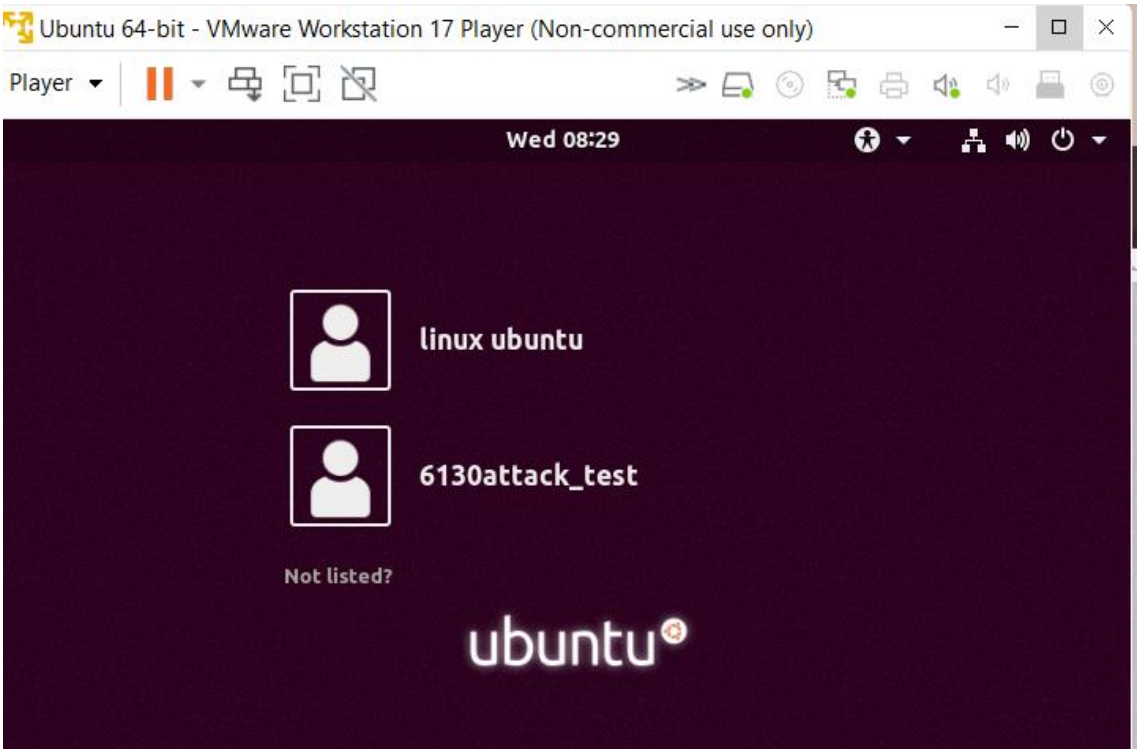


Fig 15 4.4 (3) After Attack

Abusing exposed docker registry

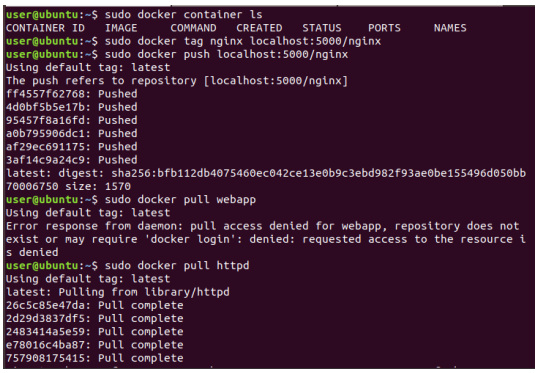


Fig 16: 4.5 (1)

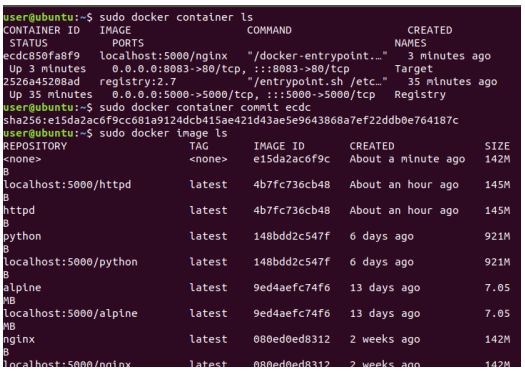


Fig 17: 4.5 (8)

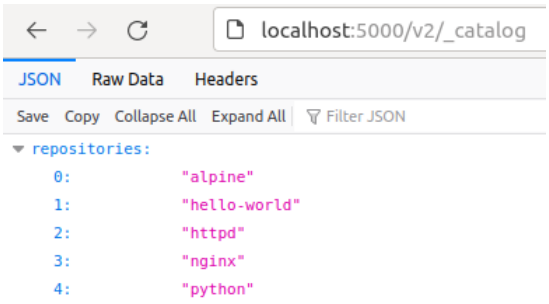


Fig 18: 4.5 (2)

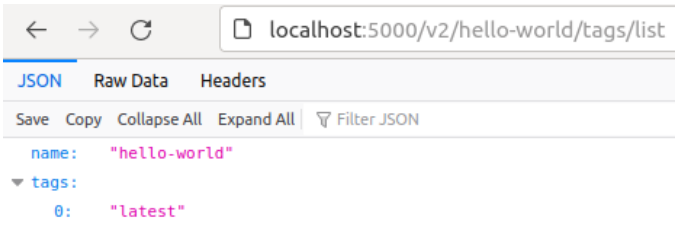


Fig 19: 4.5 (3)

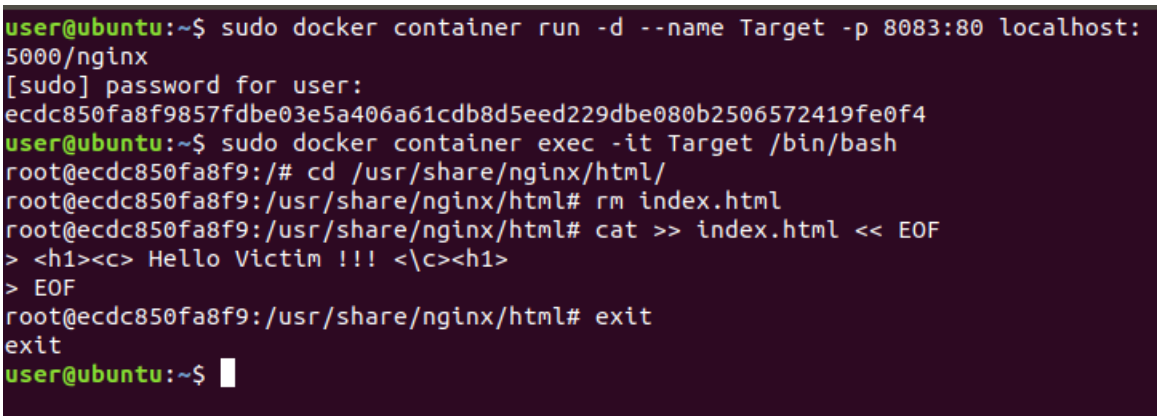


Fig 20: 4.5 (7)

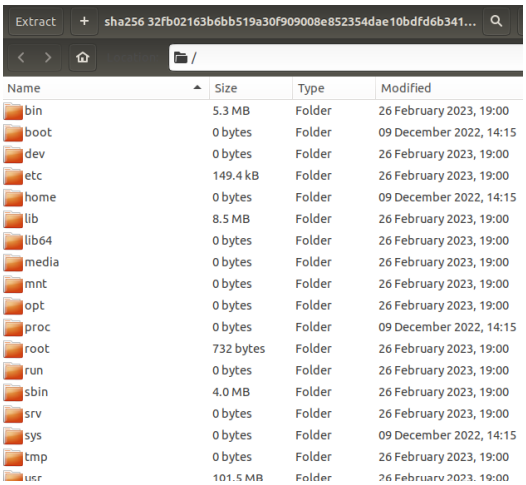


Fig 21: 4.5 (5)

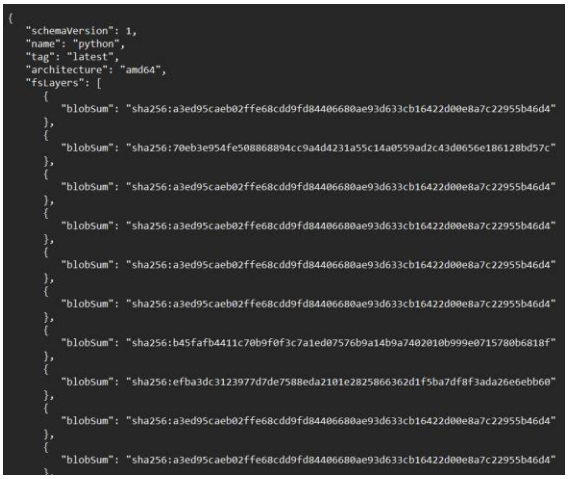


Fig 22: 4.5 (4)

Fig 23: 4.5 (6)

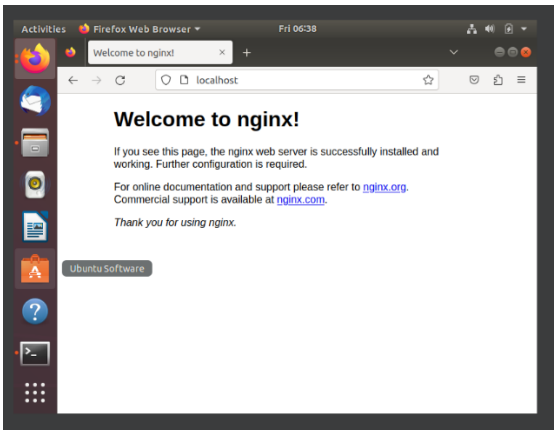


Fig 24: 4.5 (Before Attack)

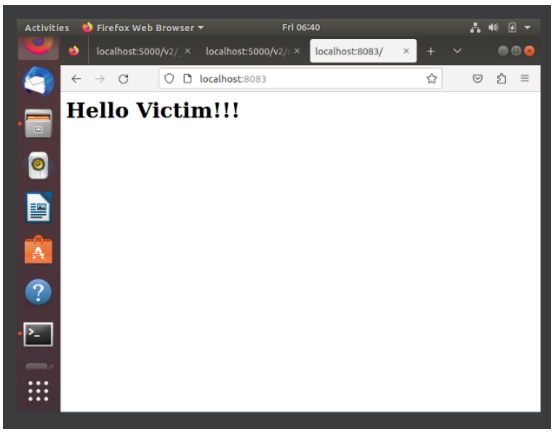


Fig 25: 4.5 (After Attack)