SECURITY AUDIT OF DOCKER CONTAINER IMAGES IN CLOUD ARCHITECTURE

Co-authored by Waheeda Syed Shameem Ahamed

Pavol Zavarsky

Bobby Swar

Project report

Submitted to the Faculty of Graduate Studies,

Concordia University of Edmonton

in Partial Fulfillment of the

Requirements for the

Final Research Project for the Degree

MASTER OF INFORMATION SYSTEMS SECURITY MANAGEMENT

Concordia University of Edmonton

FACULTY OF GRADUATE STUDIES

Edmonton, Alberta

April 2020

SECURITY AUDIT OF DOCKER CONTAINER IMAGES IN CLOUD ARCHITECTURE

Waheeda Syed Shameem Ahamed

Approved:

Pavol Zavarsky [Original Approval on File]

Pavol Zavarsky Date: April 14, 2020

Primary Supervisor

Edgar Schmidt [Original Approval on File]

Edgar Schmidt, DSocSci Date: April 27, 2020

Dean, Faculty of Graduate Studies

Security Audit of Docker Container Images in Cloud Architecture

Waheeda Syed Shameem Ahamed
Information Systems Security and Assurance Management
Concordia University of Edmonton
Edmonton, AB, Canada
wsyedsha@student.concordia.ab.ca

Pavol Zavarsky
Information Systems Security and Assurance Management
Concordia University of Edmonton
Edmonton, AB, Canada
pavol.zavarsky@concordia.ab.ca

Bobby Swar
Information Systems Security and Assurance Management
Concordia University of Edmonton
Edmonton, AB, Canada
bobby.swar@concordia.ab.ca

***Abstract*—Containers technology has radically changed the ways for packaging applications and deploying them as services in a cloud environment. According to the recent report of TrendMicro security predictions of 2020, the vulnerabilities in container components that are deployed with cloud architecture have been ranked as one of the top security concerns for development and the operations teams in enterprises. Docker is one of the leading container technologies that automate the deployment of applications into containers. Docker Hub is a public repository by Docker for storing and sharing the docker images. These Docker images are pulled from the Docker Hub repository and the security of images being used from the repositories in any cloud environment could be at risk. Vulnerabilities in docker images could have a detrimental effect on enterprise applications. In this paper, the focus is on securing the docker images using vulnerability centric approach (VCA) to detect the vulnerabilities and developing a checklist of use cases compliant with NIST standards. This paper develops a checklist of use cases to verify the standards by systematic analysis of the docker image in compliance with OWASP CSVS. The paper has the following objectives (i) to identify and assess the vulnerabilities of docker images with their CVE details using VCA; (ii) to develop a checklist of use cases compliant with the NIST guidelines for securing container images; and (iii) to align the checklist with the requirements of OWASP Container Security Verification Standards. The proposed checklist can be used as a useful tool during the development, deployment, and maintenance of the microservice application.**

Keywords—docker, containers, OWASP, cloud, vulnerabilities.

I. Introduction

Containers are transforming the enterprise applications by allowing data centers to deploy their applications rapidly with reduced development overhead, lower cost, efficient use of resources and increased business agility. Microservices are developed in container technology as the applications are broken down into smaller components and independent of services. Docker is a containerization platform which is a Platform as a Service (PaaS) product that uses the operating system virtualization to deliver software applications as packages. Docker containers are easy to deploy in cloud infrastructure and are integrated with cloud providers like AWS, Microsoft Azure, Google, and Digital Ocean. Containers have direct access to the host kernel which is considered as one of the major security weaknesses of containers security and privacy management. According to Forester's survey, 53% of the enterprises consider security as their biggest concern when developing their applications with container technology [1].

Fig 1 describes a Docker architecture workflow for creating docker images using the docker registry and the deployment of images into containers with cloud providers. Docker operates as a client-server architecture and has three major components, docker client, docker host, and docker registry. As shown in Fig 1, the docker client is the primary component for interacting with docker daemon. The client uses the commands "docker build" to build the image, "docker pull" to pull the image from the docker host and "docker run" to execute and create an image. The Docker host, which is the server, gets the request from the docker client. The Docker registry is the public repository that stores the images. As shown in Fig 1, the Docker Registry pulls the image from the Docker trusted registry of Docker which could be a private registry created by an organization or an individual in any of the public cloud providers. The public cloud providers like Amazon Elastic Container Registry, Microsoft Azure Container Registry, and Google Cloud Container Registry, store the images by fetching the images from Docker hub which is the trusted Docker registry on Cloud.

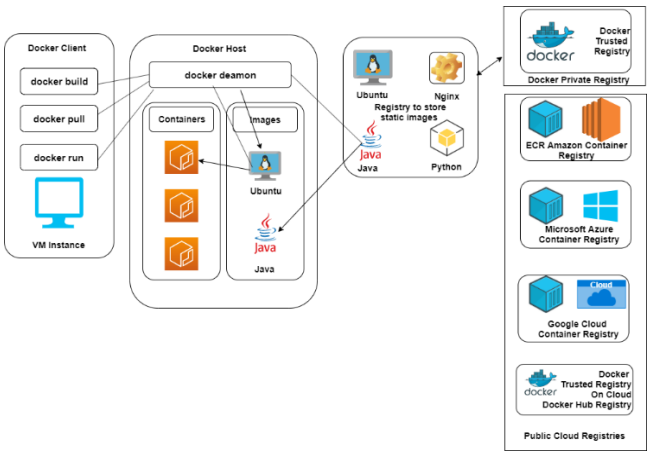

Fig 1: Docker architecture workflow

The core components of Docker are the images, registries, containers and docker hosts, of which the containers and images are high-risk components. Docker on execution searches for a simple dockerfile that defines the steps needed to generate a Docker image, which is then used to instantiate the containers [2]. Dockerfile is the configuration file that is needed to create an image and each instruction in the dockerfile creates a layer in the image. As shown in Fig 2, the blocks of image layers are the result of the instructions executed in the Dockerfile. The topmost layer of the docker image is the read/write layer and the below layers are only readable layers. The base images are the operating system images like ubuntu and alpine that form the underlying layer.

The Container Deployment's major concern is the stability and security of images. The vulnerability can be in the package version of the base image, configuration file issues, authentication, and authorization. The docker image is an initial step for developing an application in the container

technology, identifying the vulnerabilities and mitigating or preventing those issues could be achieved by the checklist of use cases developed in this research. Container Security Verification Standards (CSVS) could be used as a blueprint for the organization to create a security checklist and the main objective of this research is to develop a checklist following the NIST guidelines and to align with the requirements of OWASP Container Security Verification standards.

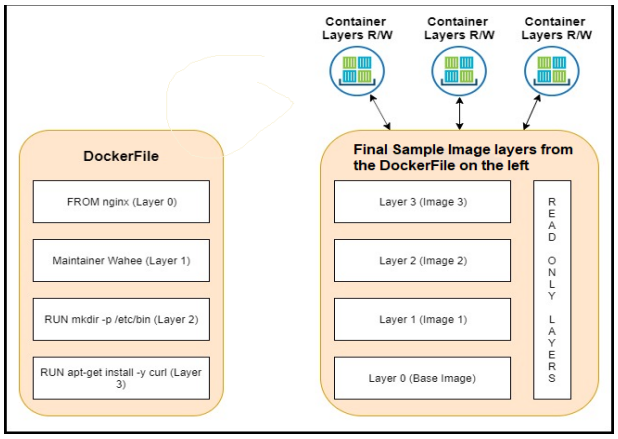

Fig 2: Container layer representation of an image

This paper is organized as follows, Section II provides the related works in the security of docker images, Section III provides the methodology followed for developing the use cases and finally, Section IV concludes the paper.

II. Related Works

National Institute of Security and Technology (NIST) defined the application container security guidelines in NIST.SP.800-190. According to NIST container security guidelines, the primary data to protect in the container technology are the images and the containers as these comprise the files and dependencies required to run an application [4]. The research in Docker image security has focused on static code analysis of images, base image security, authentication, and authorization.

Durante et al. performed a detailed analysis of Docker vulnerabilities and patches and applied a Static Code Analysis to understand how the vulnerabilities spread over time [5]. The findings indicate that image vulnerabilities could have been detected by analyzing the behavior of the software, through regression testing or robustness testing. In relation to [5], for performing the robustness and regression testing, our developed checklist could be used to verify the proper controls and configuration of an image during the development, deployment and testing phase of an application.

Zerouali et al. analyzed the relation between the outdated containers and the vulnerable packages installed in the software to enhance container deployment [6]. The images are extracted from Docker Hub and analyzed to study the packages installed in them to compute a technical lag of the image based on their package versions. After the empirical analysis of the state of the packages in Docker containers, a conclusion was derived that outdated container packages in images are one of the reasons for the presence of bugs and vulnerabilities in the docker image. It is also observed that the Debian repository from the Docker is widely used due to its maturity and widespread, as it maintains the package version for several simultaneous releases [6].

Shu et al. performed a study on the state of security vulnerabilities for both official and community images from the Docker Hub repository as they are considered as the base image for writing a Dockerfile. Shut el al. proposed a Docker Image Vulnerability Analysis (DIVA) framework to automatically discover, download and analyze the Docker images for its security vulnerabilities [7]. The experimentation analyzed a set of images and observed the following points, a) both official and community images contain an average of more than 180 vulnerabilities, b) many images had not been updated for hundreds of days, and c) vulnerabilities are commonly propagated from parent images or child images. Based on the observation by the research done, there is a need for systematic methods while analyzing the content of the Docker containers [7]. In relation to our research, the use case checklist could provide a systematic method for analyzing a Docker image during the development of an application.

Tunde et al. evaluated a set of static and dynamic attack detection schemes using 28 real-world vulnerabilities. They compared the vulnerabilities detected through these detection schemes and provided a conclusion that dynamic anomaly detection schemes improve the detection rate of vulnerabilities compared to the static vulnerability schemes [8]. For static vulnerability detection scheme Clair, an open-source tool for the analysis of static vulnerabilities in docker containers is used. For the dynamic vulnerability detection scheme, the evaluation is done with unsupervised anomaly detection schemes. The results state that dynamic anomaly detection schemes can effectively detect greater exploits with false-positive rates when compared to static vulnerability schemes [8].

OWASP Container Security Verification Standards were published in July 2019 to enhance the security of the projects deployed with containers. The Container Security Verification Standard defines security requirements or tests that can be used by architects, developers, testers, security professionals and even consumers to define a secure container with respect to its infrastructure [3]. Though there has been a lot of research on Container security related to the framework, performance, and vulnerability analysis, there is no proper standard and guidelines like a verification checklist based on standards to consider while developing a docker image. This research could help the organizations to align their applications with the OWASP Container Security Verification Standard requirements to avoid vulnerabilities in the docker image.

III. Methodology for Developing Use Cases

The aim of this research is to analyze the security of Docker images in containers and to develop a checklist of use cases to verify the security of the images based on the standards. The developed use cases are to assist the DevSecOps teams in an organization to test and verify the applications in compliance with the OWASP CSVS. Though there are integrated tools for finding the vulnerabilities in containers this checklist will be used by the DevSecOps teams in the organization while developing, testing their applications and verifying the standards that could avoid vulnerabilities in the applications. This section is divided into three sub-sections, scanning the commonly used docker images in a cloud environment and identify its vulnerabilities, developing the use cases based on the NIST guidelines, and verifying the use cases with the OWASP Container Security Verification Standards.

A. Scanning the images

In this section, we have scanned the commonly used images in five categories like operating system, database, language, web component, and application platform images. A Vulnerability Centric Approach (VCA) is applied for assessing the docker images by scanning the images and reviewing the configurations for vulnerabilities. Google Container Registry (GCR) a private container image registry that runs on Google Cloud and one of the cloud providers of Docker is used for scanning the images and it has a built-in API for scanning the image vulnerabilities, orchestration, performance measurement for the API. In Google Container Registry a Vulnerability Scanning API is enabled to automatically scan the images once it is pushed to the container registry and this could enhance the Continuous Integration/Continuous Delivery process. To scan the images in this registry the steps are as follows.

1) A set of images is pulled from the Docker Hub Registry, a public repository for Docker images into the Google Container Registry under a project created.

2) Once the images are pulled from the Docker Hub Registry repository, it is tagged with the project created in the google container registry

3) The tagged image from the above step is pushed to the specific project in the google container registry.

4) The images are automatically scanned for vulnerabilities once it is pushed to the container registry since the Vulnerability Scanning API is enabled.

5) The vulnerabilities in the images are evaluated based on the Common Vulnerability Scoring System (CVSS) and for each image, the critical and high vulnerabilities are analyzed.

From the data analyzed by scanning a set of images in each category, it is observed that the vulnerability of the images is due to package version issues, privileges given to the default user which could bypass the information to the attacker, configuration issues in Dockerfile. Considering the issues analyzed from the CVE Details it is important to harden the security of images as it could result in attacks like Denial of Service, Man in the Middle due to root privileges. To harden the security of the images, this research develops a checklist of use cases from the CVE details analyzed in the common images used in the applications.

B. Developing the Use Cases

This section is developing a checklist of use cases based on the vulnerabilities analyzed. Based on the vulnerabilities and their CVE details the docker image security use-cases have been classified into four phases.

Phase 1: Docker base image inspection (Use Cases A1 – A5)

Docker image inspection is done for the dependent images required for an application. The Docker inspect command and the Docker Host Security tool which is an audit tool could provide a detail inspection of an image in the local repository. As the base image pull command is the first command in any Dockerfile and it is mostly fetched from the Docker Hub Registry which is the repository for official images in docker, inspecting the top base images with their updated versions is essential. Trust is one of the key factors in security terms and when relying on any open-source data it is the consumer's responsibility to check whether the data is trusted. Docker Hub registry is a central repository and it depends on the publisher of the official images whether to sign the image or not. The base images need to be checked for their signed integrity before it is included in the application. To check the trust of the base images below are the use cases from A1 to A5 are to be followed.

Use Case A1 - *Check whether the image is an official image:* This can be done by the docker command "docker search <image_name> --filter is-official=true". Official images in the Docker repository are the certified and trusted images.

Use Case A2 - *Check the trust of the base image:* The images in the Docker repository may be signed or not so it is the consumer's responsibility to check the integrity of the image which is being fetched from the public repository. Docker Content Trust provides the ability to use digital signatures when the data is sent and received from the repository [9]. This can be achieved by enabling the DOCKER_CONTENT_TRUST flag to 1 in the environment variables. When this flag is enabled and if an image is not a certified or a signed image, it would show an error mentioning that the image is untrusted and won't be able to extract the image from DockerHub.

Use Case A3 - *Check the content of the image*: Docker images are comprised of a set of layers and we need to check the layers in the image to evaluate the space occupied by the image, its contents and other details. The content of the image can be checked through the docker inspect command which gives the details of network configuration, it's hash value and layer details of the image in a JSON format. The Dive tool could be used to check the unused spaces and the efficiency of a docker image [14]. This helps in uninstalling the packages which are not being used in the image as this could improve the performance of building the image.

Use Case A4 - *Uninstalling the unnecessary packages:* The size of a docker image also involves the dependent packages used to build an image. Including unnecessary dependencies in the Dockerfile increases the size of the image. It would slow down the deployment process and also increases the possibility of the attacks. One way to do this would be to include a dockerignore file which consists of a list of patterns, the CLI modifies the context to exclude files and directories matching those patterns from the dockerignore file during the build process of a docker image [15].

Use Case A5 - *Disabling the build cache*: When an image is built through Dockerfile, it steps through each instruction mentioned in the Dockerfile, executing them in chronological order. When each instruction is executed it checks the cache for any existing image layer of that instruction and if present it uses that image. But this could cause issues when there is a change in the instruction of the Dockerfile and it uses the layer form cache. Therefore, it is always preferred to create each and every layer of the image whenever the Dockerfile is executed to build.

Phase 2: Dockerfile Configurations (Use Cases B1 - B5)

When developing an application, it is the responsibility of the developer to consider and the security aspects of an application. Configuration issues are the potential ways to avoid possible security threats through Dockerfile. Rapid development cycles reduce the focus of security and vulnerability testing to be done by the developers. Below are the configurations to consider while writing a Dockerfile in rapid development as recommended in [10]

TABLE I. USECASE ID WITH DESCRIPTION AND MAPPING TO DEVSECOPS

Phase	Use Case ID	Description	DevSecOps
Docker Base Image Inspection	A1	Check whether the image is an official image	Development, Security, Operations
	A2	Check the trust of the base image	Operations, Security
	A3	Check the content of the image	Security
	A4	Uninstalling the unnecessary packages	Operations
	A5	Disabling the build cache	Operations
Dockerfile Configurations	B1	Base image version	Development
	B2	Running apt-get install and apt-get update	Development
	B3	Using COPY instead of ADD	Development
	B4	HealthCheck for container images	Development, Operations
	B5	Secrets not stored in dockerfile	Development
Image Authentication	C1	Image signing with DCT	Development, Security
	C2	Registry Authentication	Development, Security
Image Authorization	D1	Creating a user for each container	Development and Operations
	D2	Permissions for user Id is removed	Development, Security, Operations

Use Case B1 - *Base image version:* The base image version is one of the reasons for the vulnerabilities. The usual ways to include the base image in the docker file are through the latest tag or through a version tag. When using the latest tag it violates the immutability of the image as it is frequently updated and changes the size and other features of an image. When using the specific version tag, for example, 1.0, it is just a name given by the publisher for that image and it may change or delete in the future. The best way to pull the image while mentioning in the Dockerfile is through hash value, the digest of an image. The image ID of an image is a SHA256 hash value which is unique to each image. When the image tag changes or the latest tag gets updated, a new SHA256 is created but the SHA256 remains the same for the image which is considered as the immutable identifier for the image.

Use Case B2 - *Running apt-get install and apt-get update*: In Linux terminology, the Advanced Package Tool (apt) is a command-line tool to interact with the package system. In docker, the packages are installed and updated using this tool. When writing the Dockerfile the apt-get command is used with the RUN command in the docker file and it can be with multiple arguments. The apt-get install and apt-get update methods are mostly used in the Docker file when the packages need to be installed are updated with the RUN command [10]. There are two ways to install and update the packages through Dockerfile.

apt-get update and apt-get install in a separate line: When the image is built all the packages are updated with the latest version, but it loses the immutability of an image [11].

apt-get update and apt-get install in the same lines: When the image is built it ensures that there is no intervention between the execution process as the cache is executed after all the packages are installed which is mentioned in the RUN command [11].

Use Case B3 - *Using COPY instead of ADD:* The commands which create the layers in the image are RUN, ADD and COPY command. Other commands create the intermediate layers but not the writeable layers.

ADD: This command copies the files from source to the destination directory, and if the destination directory is not available it creates it implicitly. It is also used to fetch data from remote URL's and it could result in the Man-in-the-middle attack when there is a need to download data directly into a secure location and the attacker can modify the content of the file being downloaded [12]. The advantage of using ADD is when we need to extract any archive files or zipped files that are used. But this could also provide a risk for vulnerabilities in the archive files and most of the vulnerabilities in the images are through their inbuilt packages and libraries.

COPY: When using COPY, it does not create the directory or file implicitly, and when the file is copied from a remote URL, it is connected with a secure TLS connection and the source is validated [12].In COPY the files are copied from local folders and from the host. When an external file is copied it needs to be authenticated, which is secured and the preferred method for file transfer.

Use Case B4 - *HealthCheck for Container Images:* Healthcheck for container images is to determine the state of the services whether they are running or not. In this healthcheck, we can specify the interval time on how frequently the state of the container has to be checked. When the healthcheck is enabled, the container can have one of the three states, a) Starting - when the container is starting, b) Healthy - when the healthcheck command in the docker file executes successfully and the container is up in running state, and c) Unhealthy - If the container takes a long time to start, then we can also specify the timeout interval in the healthcheck command which will give the timeout error after a specified time.

Use Case B5 - *Secrets not stored in Dockerfile*: A docker image is shared in different infrastructures and designed to run in any environment. If any of the packages need credentials it has to be passed in the runtime and it not be built into an image. When a Dockerfile contains any hardcoded secrets or configuration values it is not encrypted. A secret can be a certificate, SSH key, password which consists of a blog of data. Docker secrets can be created and invoked during initializing a container. The secrets are created and sent to the Docker engine through the TLS connection.

Phase 3: Image Authentication (Use Cases C1 - C2)

The integrity of an image assures that the image is not changed when it is pulled from the repository and not from any third-party resources. Below are the scenarios through which the integrity of the image can be assured.

Use Case C1 - *Image signing with DCT*: Docker Content Trust is a feature of Docker which is used to verify the client-side or runtime verification for the integrity of an image, and

the publisher of specific image tags through digital signature. An image can also be signed and pushed to the Docker registry with the docker trust command and it is built on top of the Notary feature in Docker. For signing an image, the Docker registry with the attached Notary server is required. Notary in Docker is a collection of trusted content that includes the sign collections of the consumers and the publishers of the images. The trust collections in Notary is used with Globally Unique Names (GNU). It is responsible for the operations necessary to create, manage, and distribute the metadata of the images to ensure the integrity and the freshness of the content of the image.

Use Case C2 - *Registry Authentication:* Registry stores the container images and it should be authenticated before being accessed by a user. DockerHub provides the option of creating private registries for the organizations where they can store their internal images. This registry is integrated with the cloud environment like Google Container Registry so that when there is a need to integrate our code with any other tool it can be done with ease. Authentication is an important step to consider in the process as the images are stored in the registry and if an illegitimate user gets accessed to the registry, there is a risk of exploiting the image with malicious content. If the image is configured as private, the registry access should be given to the users who are working on the applications. The effective way to authenticate the registry is to provide the account credentials through encrypted dockercfg config file. This file secures your credentials when we push or pull the images from private registries.

Phase 4: Image Authorization (Use Cases D1 - D2)

Use Case D1 - *Creating a user for each container:* Images which are created as container should not be created with a root user as this might cause access restriction bypass by any external users. This is to harden the container and securing the container with restricted access. By default when base image is pulled from the Docker Hub in the FROM command of the Dockerfile it is by default given the root privileges. Containers provide isolation, but it has a direct interaction with the host kernel, from the underlying operating system. To avoid the potential risk, containers should always run under non-privileged user. Users can be created based on groups working on the application or an individual user for each container.

Use Case D2 - *Permissions for user Id is removed:* The user Id is referred to as UID and its managed by the Linux kernel. The kernel system calls to determine whether the requested user has privileges to access the container. This is achieved by creating a user under a group and providing the privileges for the specific user to run the container. The unwanted permission for a user should be removed to prevent the escalation attacks, which means if an illegitimate user can access the container resources with UID, it can provide access to other containers on the host resulting in the Denial of Service attack.

C. Verifying the use cases with OWSAP CSVS.

The third section is to verify the standards of OWASP Container Security Verification Standards and to assign which role of users can use the use cases to verify the standards in their workflow process. Tailoring the CSVS with the use cases will increase the focus on the security requirements that are most important for the projects and environments in organizations. The OWASP Container Security Verification Standards is to establish a framework of security requirements and controls that focus on normalizing the functional and non-functional security controls required when designing, developing and testing container-based solutions [3]. The main goals of CSVS are a) help organizations to maintain and secure container infrastructure, and b) allow security services and consumers to align their requirements and offerings [3]. A checklist of use cases has been developed based on the critical vulnerabilities in the images. In OWASP there are 3 levels of Container Security Verification Security Standards as mention in the guidelines of the standards.

Level 1: This is the minimum requirement of security needed for all containers. A container-based infrastructure achieves the CSVS level 1 if it adequately defends against well-known security threats that are easy to discover and easy to abuse [3].

Level 2: This is for the projects which handle sensitive information. A container-based infrastructure achieves this level when it adequately defends against most of the risks associated with the containers. This level is imperative especially for a financial business organization as they implement business-critical and sensitive functions [3].

Level 3: This is the highest level of the verification standard. This is for the container-based solutions which require significant levels of security verifications, such as in military defends, health and safety, critical infrastructure [3]. The failure to achieve this level in the organization could result in a significant impact on the organization's operations.

Therefore, the checklist developed int this research is used to explain how the verification of OWASP Container Security Verification Standards can be done, and each verification rule is mapped with the use case as shown in Appendix B. The organizations could be benefited by this use case checklist during the development, testing and audit of an application.

IV. CONCLUSION

In this paper, critical vulnerabilities of Docker images are identified, and a checklist of use cases is developed to ensure the security of Docker images by the development, operations and security teams of an organization. In addition to that, the main objective achieved could be to verify the OWASP Container Security Verification Standards with the developed use-cases checklist, which can be used by the enterprises for increasing the focus of security requirements on their projects. The checklist could be extended for vulnerabilities in the orchestration of containers which is important for container interactions.

REFERENCES

[1] J. Shetty, “A State-of-Art Review of Docker Container Security Issues and Solutions”. American International Journal of Research in Science, Technology, Engineering & Mathematics, 2017.

[2] T. Yarygina, A. Bagge, “Overcoming Security Challenges in Microservice Architectures”. 11-20. 10.1109/SOSE.2018.00011, 2018.

[3] OWASP, “OWASP Container Security Verification Standards”, July 2019.

[4] M. Souppaya, J.Morello, K. Scarfone, “Application Container Security Guide”, NIST Special Publication 800-190, September 2017.

[5] A. Duarte, N. Antunes“An Empirical Study of Docker Vulnerabilities and of Static Code Analysis Applicability”. 27-36. 10.1109/LADC.2018.00013, 2018

[6] A. Zerouali, T. Mens, G. Robles, Jesus M.Gouzalez-Barohana, “On the Relation Between Outdated Docker Containers, Severity Vulnerabilities, and Bugs”, 2018.

[7] R. Shu, X. Gu, and W. Enck, “A Study of Security Vulnerabilities on Docker Hub”, *7th ACM Conference on Data and Application Security and Privacy*. ACM, 2017, pp. 269–280.

[8] O. Tunde-Onadele, J. He, T. Dai and X. Gu, "A Study on Container Vulnerability Exploit Detection," *2019 IEEE International Conference on Cloud Engineering (IC2E)*, Prague, Czech Republic, 2019, pp. 121-127. DOI: 10.1109/IC2E.2019.00026.

[9] “Content Trust in Docker”, Available at: https://docs.docker.com/engine/security/trust/content_trust/.

[10] Jorge Silva, “9 Common Mistakes in Dockerfile”, runnable, Available at : https://runnable.com/blog/9-common-dockerfile-mistakes.

[11] Docker Documentation, “Best Practices for writing Dockerfile”. Available at:https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

[12] Liran Tal, “Why you should use COPY instead of ADD when building Docker image”.Available at: https://dev.to/lirantal/why-you-should-use-copy-instead-of-add-when-building-docker-images-5fkj

[13] “The New Norm: TrendMicro Security Predictions for 2020”, TrendMicro,Available at :https://www.trendmicro.com/vinfo/us/security/research-and-analysis/predictions/2020.

[14] “How to analyze and explore the contents of docker images”, October 2019, Available at:https://www.ostechnix.com/how-to-analyze-and-explore-the-contents-of-docker-images/.

[15] “DockerFile Reference”, Available at: “https://docs.docker.com/engine/reference/builder/#dockerignore-file”

APPENDIX A

This appendix is divided into two sections. The first section provides the results of the images being scanned through vulnerability centric approach in Google Container Registry (GCR). The second section provides the results of each use case developed in section III of this paper.

A. Scanning the images in GCR

The images are pulled from the docker repository and pushed to the GCR repository created with a project name. The below screenshot shows the project id being set gcloud registry and the images being pulled and pushed to the gcloud registry which is the google cloud registry.

Fig 3: Setting the project id for the google cloud

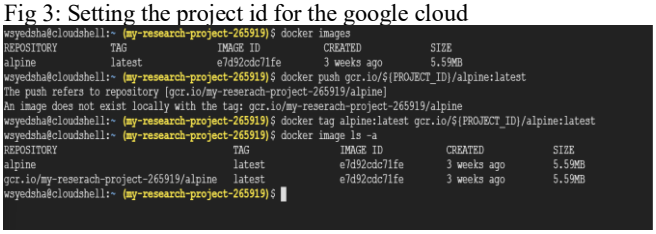

Fig 4: Docker images being tagged and pushed to the google cloud container registry

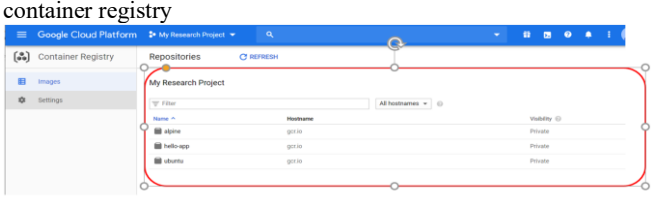

Fig 5: List of images in the google cloud container registry

Once the images are pushed to the registry it automatically scans for the vulnerabilities in the images using the Vulnerability Scanning API.

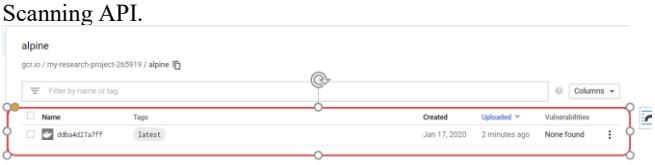

Fig 6: The latest version of the image without vulnerabilities.

Fig 7: An older version of the image which has vulnerabilities.

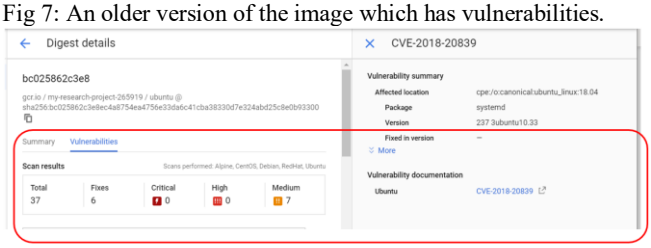

Fig 8: Vulnerabilities based on severity index.

B. Results of the Developed Use Case

This section is to provide the result of each use case identified and tested. A project is created in the Google container registry to do the experimentations. All the images we pushed to the registry will be stored in the gcr.io/[PROJECT_ID] and when we list all the images in our google container registry are listed as shown in the below screenshot.

Fig 9: List of images from google container registry in the project.

Use Case A1 - *Check whether the image is an official image*

This is to check whether the dependent images we use in the Dockerfile are the official images. The is-official filter is added to check the official and unofficial images by setting the value as true and false. The below screenshot shows the list of official and unofficial Ubuntu images from the docker registry.

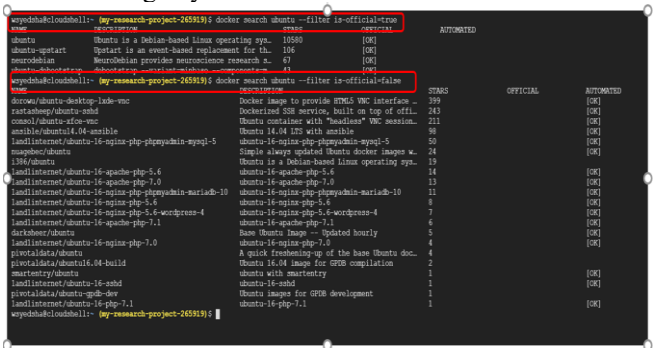

Fig 10: Listing the official and unofficial open-source images.

Use Case A2 - *Check the trust of the base image*

The trust of the base images is checked by enabling the variable DOCKER_CONTENT_TRUST value as 1 in the environment variables. The below screenshots show when the variable is set to one and an older version of the ubuntu image is pulled from the docker registry it shows an error as “No valid trust data for 16.01” which means no trust for the specified image. Whereas when we pull the updated and the

latest image from the repository it is being fetched from the repository.

Fig 11: Enabling Docker Content Trust and trying to pull a trusted and untrusted image from the repository.

Use Case A3 - *Check the content of the image*

To check the content of an image we can use docker inspect which shows the configuration details of an image. A Dive is an open-source tool that allows to traverse through the packages and also specifies the unused spaces being used by a specific package. This helps in uninstalling the unnecessary packages in the next use case. The below screenshot shows the details of an ubuntu image in which a dive tool shows the efficiency and unused space of an image.

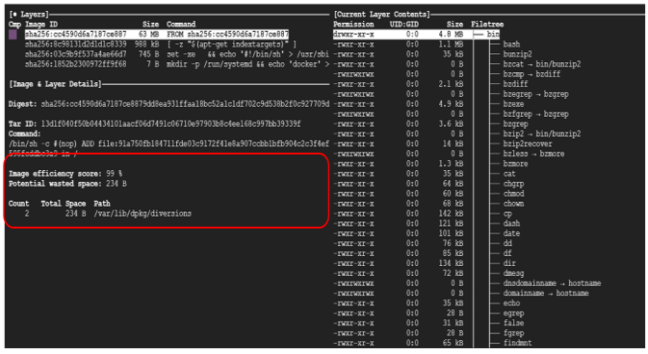

Fig 12. Analyzing the content of a docker image.

Use Case A4 - *Uninstalling the unnecessary packages*

In a docker image, the unnecessary packages are uninstalled to reduce the complexity in the image, which also reduces the chances of vulnerability and exploits. The dockerignore file is placed in the root directory and contains a list of patterns, the CLI modifies the context to exclude files and directories matching those patterns from the dockerignore file [15]. It is one of the ways to optimize the docker image. Below is the sample dockerignore file to remove unnecessary files and packages while building a docker image.

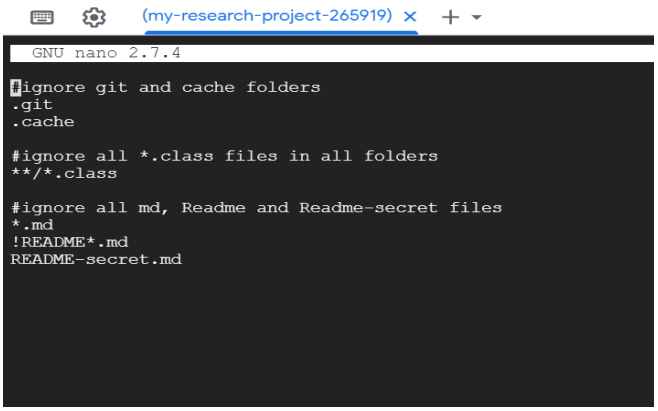

Fig 13: A sample Dockerignore file to avoid including the sensitive and unnecessary files to reduce the image size and avoid complexity.

Use Case A5 - *Disabling the build cache*

When each instruction is executed it checks the cache for any existing image layer of that instruction and if present it uses that image. But this could cause issues when there is a change in the instruction of the Dockerfile and it uses the layer form cache. Therefore, when we build the image from a Dockerfile a no-cache tag fetches the latest version from the repository and does not fetch from the cache. The below screenshot shows the image pulled without cache and with the cache.

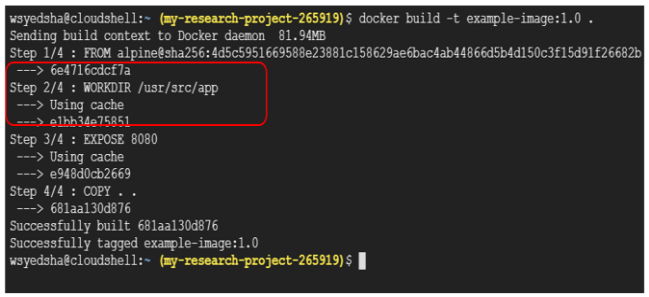

Fig 14: A build that uses the cache from its previous build of an image.

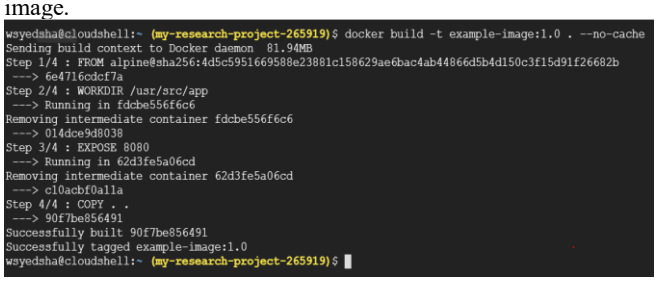

Fig 15: A build that does not use cache and keeps the packages up to date to avoid vulnerabilities.

Use Case B1 - *Base image version*

The image version may also lead to vulnerability in the applications. When using the latest tag, it violates the immutability of the image as it is frequently updated and changes the size and other features of an image. The best way to pull the image while mentioning in the Dockerfile is through hash value, *the* digest of *an* image. The below screenshot shows how the image is pulled when we mention a Sha256 value in the Dockerfile to pull an image.

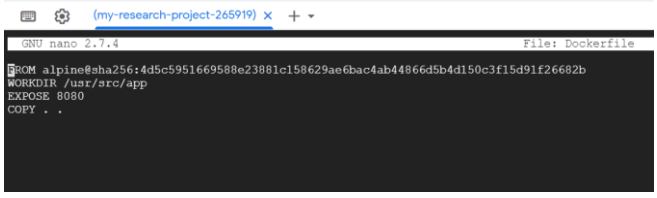

Fig 16: A sample Docker file that uses the digest of an image to be immutable at any stage of its build.

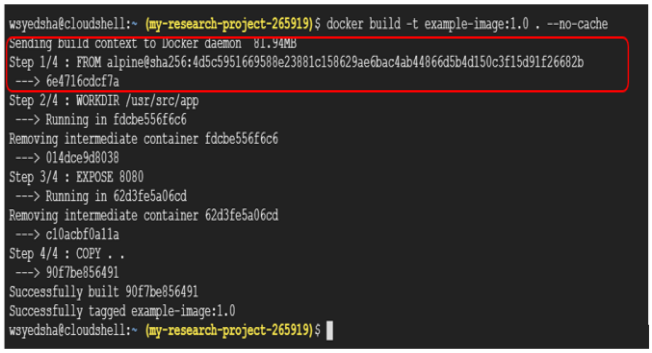

Fig 17: A build of an image with SHA256 digest of an image

Use Case B2 - *Running apt-get install and apt-get update*

When executing the apt-get update and apt-get install as separate instructions, it fetches the image layers form the cache and the changes are not reflected. To avoid this it is recommended to create instructions for apt-get install and apt-get update as a single instruction as there would not be any intervention in between for the cache and the changes also would be executed and the new layer is created for the changed instruction in the Dockerfile

Fig 18: Using apt-get update and apt-get install together to avoid cache.

Use Case B3 - *Using COPY instead of ADD*

Using the COPY command instead of ADD is to ensure that the files being copied from external sources are authenticated for the secure transfer. ADD command does not provide the authentication when a file is copied from a local or remote location.

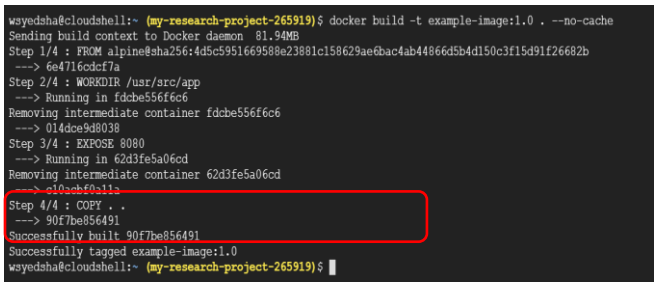

Fig 19: An image build from Dockerfile which uses COPY instead of ADD.

Use Case B4 - *HealthCheck for Container Images*

The Health check command is used to monitor the status of the container when the image is built and run as a container. When the health check is mentioned in the Dockerfile only then it will be enabled for the container.

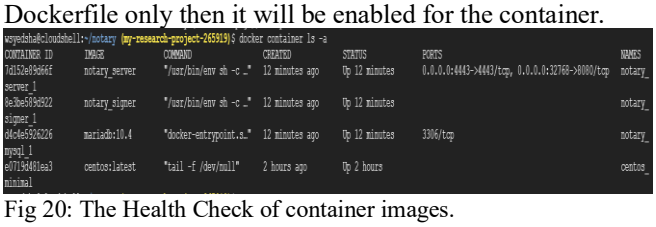

Fig 20: The Health Check of container images.

Use Case B5 - *Secrets not stored in Dockerfile*

The secrets refer to the password or any configuration information required to build an application from a docker image. These details could be passed during runtime while the image is executed instead of storing in Dockerfile which is vulnerable to attack through improper access controls. The below screenshots show how a secret can be created and passed to the image during its build.

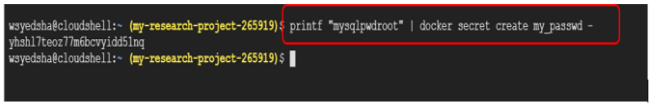

Fig 21: Creating a password to be passed for the MySQL database image during its build operation.

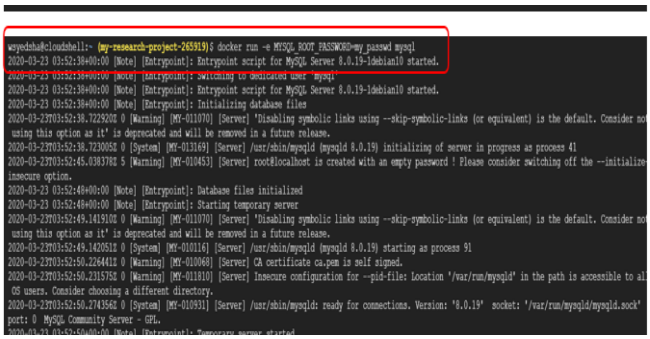

Fig 22: A MySQL image build and using the secret in runtime which was created.

Use Case C1 - *Image signing with DCT*

The image signing is to maintain the integrity of the image when we are pushing our image, or we pull an image from a public repository. The below screenshot shows an image which pulled and inspected for its integrity.

Fig 23: An image pulled from a user repository and inspected for its trust

As it is noted form the above image there is no signature or trust for that image. So, we can create a sign of the image with a trusted sig key we have created. We have created a trusted key trsutkey.pub and the image is going to be signed with a secured key by a passphrase so that whoever tries to pull the image from the public repository needs to know the signing key to pull the image.

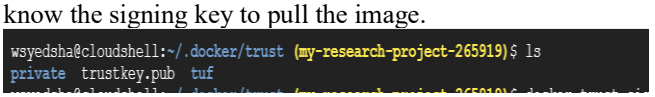

Fig 24: A signing key created with trustkey.pub

Fig 25: The image signed with the trust key

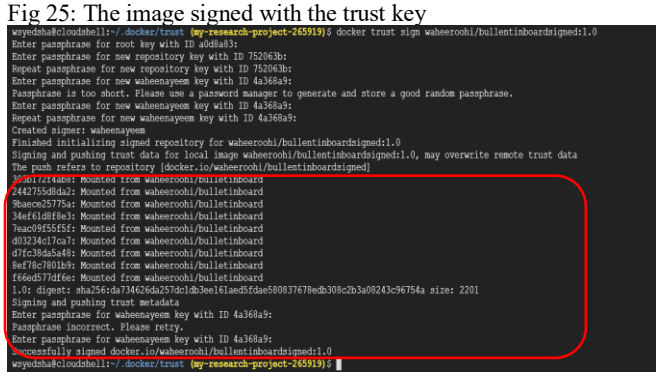

Fig 26: Signing the image and pushing with the private key into the repository

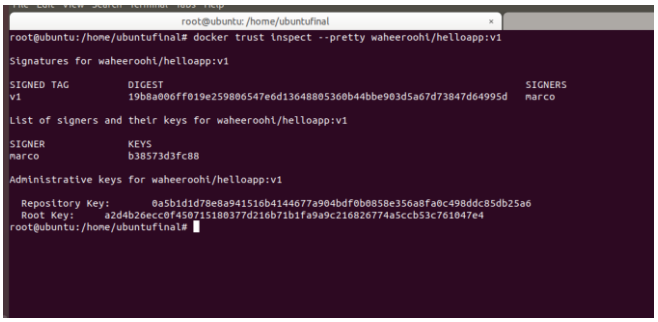

Fig 27: Inspecting the image to check it is signed with the right key.

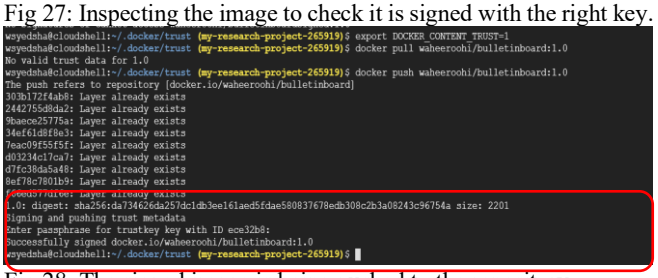

Fig 28: The signed image is being pushed to the repository.

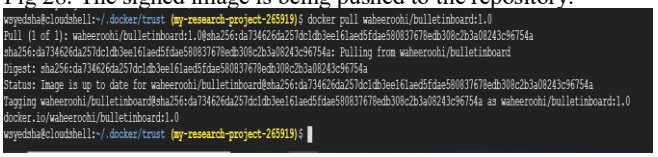

Fig 29: The image is pulled again from the repository and the SHA256 Digest key is compared.

Use Case C2 - *Registry Authentication*

The registry authentication is required to provide access only to specific users to access the image. In an organization, a group of users would be given access to a project registry in which they are working, and the other users won't be able to access the images from the registry.

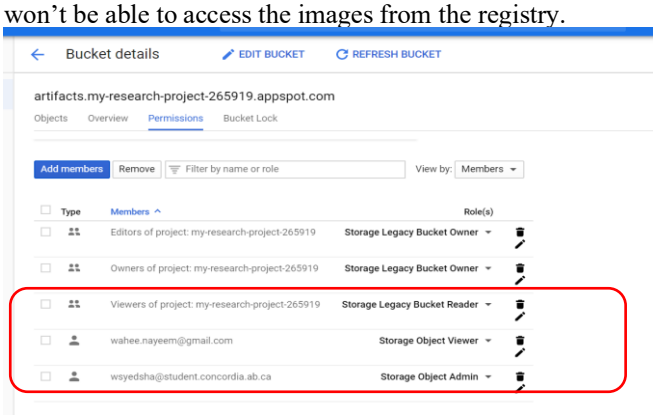

Fig 29: Admin User and Object viewer email id assigned for the project authentication.

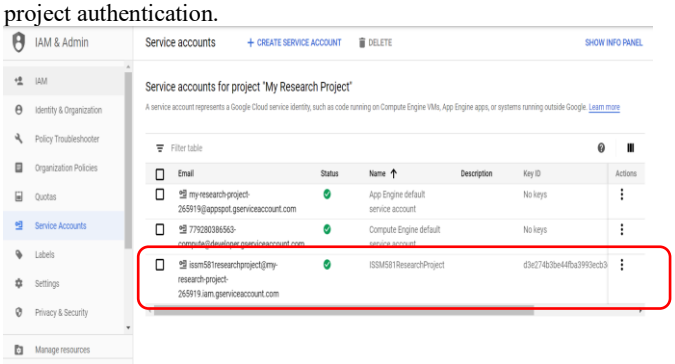

Fig 29: A private Key assigned for the project registry.

Use Case D1 - *Creating a user for each container*

This is to create a user for each container for authorizing the services. The below screenshot shows the project details, its members assigned for each role.

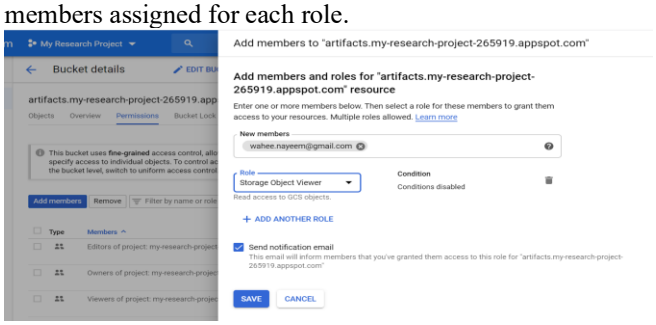

Fig 30: Adding members with specific roles to the project

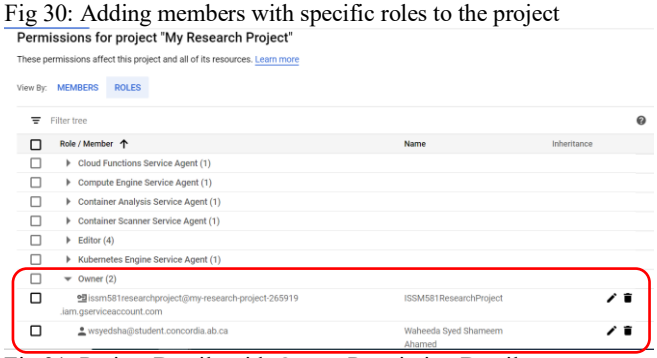

Fig 31: Project Details with Owner Permission Details

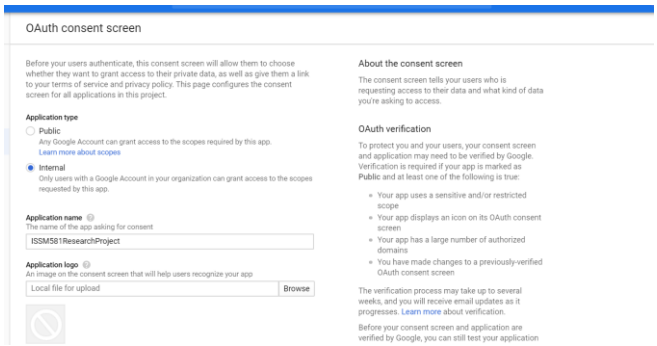

Fig 32: Authentication Consent for the project

Use Case D2 - *Permissions for user Id is removed*

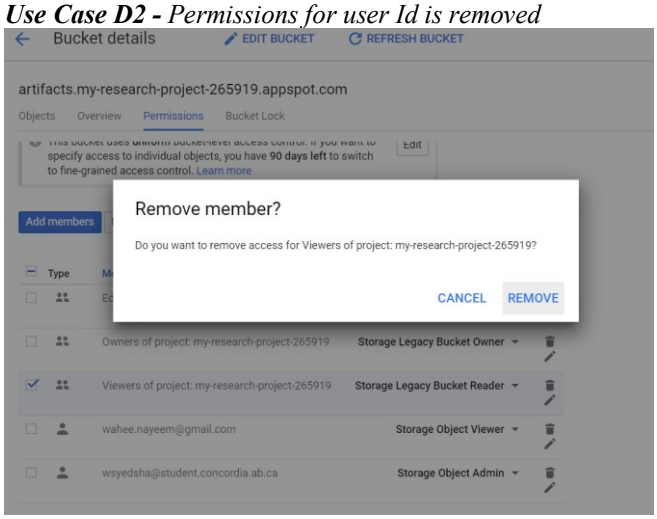

Fig 33: Remove the user from the project if not needed.

APPENDIX B: MAPPING OF THE DEVELOPED USECASES TO THE REQUIREMENTS OF CONTAINER SECURITY VERIFICATION STANDARDS (CSVS)

This Appendix section describes the verifications done for the OWASP Container Security Verification Standard with the developed use cases. The OWASP CSVS are created based on the Application Container Security Guidelines by NIST for securing the containers from host to the daemon. The verifications of OWASP CSVS are the verification rules of CSVS standards [3]. Below are the details of the column in the Appendix table.

Verification of OWASP CSVS: The verification to be done for Docker images as per the OWASP Container Security Verification Standards.

Levels of OWASP: The level of security needed for each use case

L1: A basic requirement of security needed for all containers.

L2: Requirements for the containers which are developed for sensitive business logic that requires additional protection and ensures whether the security levels are in place and tested.

L3: The highest level of verification that requires significant levels of security verifications, which are appropriate for processes in defense departments, health and safety departments, etc.

Verification of OWASP CSVS	*Levels of OWASP CSVS*			*Use Case ID*													
	L1	*L2*	*L3*	*A1*	*A2*	*A3*	*A4*	*A5*	*B1*	*B2*	*B3*	*B4*	*B5*	*C1*	*C2*	*D1*	*D2*
Verify that within each container image, a new user is created, which is then used to perform all operations within the containers.			✔													✔	
Verify that all base images are explicitly specified, using their hash instead of name and tag.			✔						✔								
Verify that the signature of each image is verified before productive usage.			✔		✔									✔			
Verify that only required software packages are installed in the images.	✔	✔	✔				✔										
Verify the Dockerfile uses the COPY directive instead of the ADD directive unless the source is fully trusted.	✔	✔	✔								✔						
Verify that Docker's health checking functionality is used for all containers and their status is monitored.		✔	✔									✔					
Verify that garbage collection is enabled on image registries and running on a regular basis.																	
Verify that after a container has been actively accessed (e.g., for troubleshooting), it's deleted and replaced by a new instance (container) of the image		✔	✔														
Verify Docker Content Trust is enabled and enforced.					✔									✔			
Verify that Dockerfile or Docker-Compose file does not contain any sensitive information like API keys and passwords.													✔				
Verify that an odd number of image registries (e.g., DTR) with a minimum of three registries is used.			✔														
Verify that containers are always created based on the most recent corresponding image and not local caches.	✔	✔	✔					✔		✔							
Verify that all images are using tags whereas only production/master is allowed to use the default latest tag.		✔	✔						✔								
Verify that the CI/CD tools and systems are connected to the Docker infrastructure to enable changes in nodes, images, or the network to be tested and rolled out fully automated.		✔	✔														