UNIVERSITY OF NAIROBI

SCHOOL OF COMPUTING AND INFORMATICS

ACCESS CONTROL MODEL FOR CONTAINER-BASED VIRTUAL ENVIRONMENTS

BY:

Titus Murithi Rugendo

P53/10900/2018

SUPERVISOR:

Dr. Andrew M. Kahonge

A project submitted in partial fulfillment of requirements for the award of the Degree of Master of Science in Distributed Computing Technology of University of Nairobi 2021.

DECLARATION

I hereby declare that the work submitted is my own work and has not been submitted at this or any other institution.

Signature .... Date .....27/08/2020....

Titus Murithi Rugendo
P53/10900/2018

This project report has been submitted for examination with my approval as University Supervisor:

Signature .... Date.....28/08/2020.....

Dr. Andrew M. Kahonge

DEDICATION

I dedicate this Project to my dear mother Rose Keeru for her unending support in my education pursuits.

To my dad Gilbert Rugendo for his great sacrifice throughout my life.

To my sister Esther Kagendo and my brother Eric Mwenda for their great support and encouragement.

ACKNOWLEDGMENTS

I wish to acknowledge and thank the Almighty God for giving me the time, knowledge and strength, enabling me to complete my project.

I wish to express my sincere gratitude to my supervisor Dr. Andrew Mwaura for your great support, time, invaluable advice and guidance that you provided to make this project a success.

I also wish to thank all the members of the project panel Dr. Andrew Kahonge, Dr. Elisha Abade, Mr. Eric Ayienga, Prof. Timothy Waema and Ms. Selina Ochukut for their dedicated guidance, motivation and encouragement during the research period. I also thank other members of the faculty for playing a great role in supporting and guiding me over the years.

Finally, I would like to thank my friends Pascal Mutulu and Joseph Mwamba for their encouragements and for always challenging me during the research period. This helped to keep me focused all the time.

ABSTRACT

With rapid development and adoption of virtualization technology, security concerns have become more prominent. Access control is the focal point when it comes to security. Since, it determines if a user can access a system and perform the action they intend to. Containers provide an all or nothing access control mechanism. Where if a host machine user has privileged access then they can access the containers as super users, enabling them to perform any desired action. All unprivileged users on the host machine are denied access to the container environment.

This research focuses on the concept of access control in container environment. It is geared more towards Docker container environment since it is the most widely adopted containerization technology. The study also focuses on analyzing existing container authorization plugins to determine how they implement access control or other forms of authorizations in containers. This research is based on exploratory research design. Since, it will involve numerous tests to determine how container engines interact with third party plugins to extend container functionalities. And, how current container authorization plugins implement access control.

Additionally, this research led to the development of a container access plugin for Docker containers that deny all host system users access to the containers. Access to container environments is determined based on users created within the containers. A container user access rights are checked against a defined access policy to determine if a user can access the container. Users allowed access are logged in as non-root users, with read privileges only. The administrator is responsible for allocating different rights and privileges to container users, this limits actions a user can perform within the containers. Thus, to achieve access control in container-based virtual environments the administrators must create and use container users to determine who has access to specific containers and the roles they can perform within the container environment.

TABLE OF CONTENTS

DECLARATION I

DEDICATION II

ACKNOWLEDGMENTS III

ABSTRACT IV

TABLE OF CONTENTS V

LIST OF FIGURES VIII

LIST OF TABLES IX

ABBREVIATIONS X

CHAPTER 1 1

1.0 INTRODUCTION 1

1.1 BACKGROUND 1
1.2 PROBLEM STATEMENT 3
1.3 OBJECTIVES 4
1.4 RESEARCH QUESTIONS 4
1.5 SIGNIFICANCE 4
1.6 JUSTIFICATION 4

CHAPTER 2 6

2.0 LITERATURE REVIEW 6

2.1 CONTAINERIZATION 6
2.2 LXC AND LXD 6
2.3 DOCKER CONTAINERS OVERVIEW 7
2.3.1 Docker Security 8

2.4 ACCESS CONTROL 8
2.5 GRANULARITY 9
2.5.1 Attribute Based Access Control (ABAC) *9*
2.6 RELATED WORKS 11
2.6.1 AuthZ Plugin *11*
2.6.2 Distributed Role-Based Access Control (DRBAC) *12*
2.6.3 Open Policy Agent Docker Authorization Plugin (OPA) *13*
2.6.4 docker-authz-plugin by Everett Toews *15*
2.7 DOCKER ADMINISTRATION 15
2.8 CONCEPTUAL MODEL 16
CHAPTER 3 17
3.0 RESEARCH METHODOLOGY 17
3.1 RESEARCH DESIGN 17
3.2 POPULATION 18
3.3 SAMPLE SIZE 18
3.4 DATA COLLECTION 19
3.4.1 Focus Groups *19*
3.4.2 Observations *19*
3.5 DATA ANALYSIS 19
3.6 PLUGIN DEVELOPMENT 20
3.7 TESTING THE PLUGIN 20
3.8 TOOLS 21
CHAPTER 4 22
4.0 RESULTS AND DISCUSSIONS 22
4.1 RESULTS 22
4.1.1 Selected Sample Size *22*
4.1.2 Focus group data *22*
4.1.3 Testing and Observations *23*
4.1.4 Analysis of collected data *24*
4.1.5 Plugin Development *26*
4.1.6 Plugin Testing *28*
4.1.6.1 Unit testing *28*

4.1.6.2 System testing *28*
4.1.6.3 Usability testing *40*
4.2 DISCUSSION 40
CHAPTER 5 44
5.0 CONCLUSION AND RECOMMENDATIONS 44
5.1 CONCLUSION 44
5.2 LIMITATIONS FOR THIS STUDY 45
5.3 RECOMMENDATIONS FOR FURTHER WORK 46
REFERENCES 47
APPENDICES 49
PROJECT SCHEDULE 49
SAMPLE CODE: 49
GUIDE ON HOW TO USE THE CONTAINER ACCESS CONTROL PLUGIN (DOCKER-AUTHZ): ...50

LIST OF FIGURES

Figure 1: Container-based Virtualization Architecture (Sultan, Ahmad, & Dimitriou, 2019) 2
Figure 2: Hypervisor-based Virtualization Architecture (Sultan, Ahmad, & Dimitriou, 2019). 2
Figure 3: EBAC Methodology (Good, 2018). 11
Figure 4: Authorization allow scenario (Levin, 2016). 12
Figure 5: Authorization deny scenario (Levin, 2016) 12
Figure 6: Open Policy Agent Architecture (Block & Spazzoli, 2019) 14
Figure 7: Conceptual Model 16
Figure 8: Research Design Method 18
Figure 9: Plugin major components Development 27
Figure 10: How Plugin is Discovered by Container Engine 27
Figure 11: Plugin Information State Module Test 28
Figure 12: Initial Policy definition 29
Figure 13: Host users access test 30
Figure 14: Container Users access Test 31
Figure 15: Container Users Privileges 32
Figure 16: User titus creating a file in non-root mode 32
Figure 17: User titus request denied 33
Figure 18: Unprivileged user rugendo request denied 33
Figure 19:File creation and editing in root mode 34
Figure 20: Changing policy file ownership 35
Figure 21: Denying all other groups all privileges 36
Figure 22: Host users' groups 36
Figure 23: Policy file read test 37
Figure 24: Policy editing 37
Figure 25: New Policy Test 38
Figure 26: Default container root user access request 39
Figure 27: Non-Container user test 39
Figure 28: Usability test Response 40

LIST OF TABLES

Table 1: Content Analysis 25
Table 2: Framework Analysis 26

ABBREVIATIONS

LXC	Linux Containers
OpenVZ	Open Virtuozzo
OS	Operating System
RTK	Rocket
API	Application Programming Interface
ABAC	Attribute Based Access Control
DRBAC	Distributed Role-Based Access Control
HTTP	HyperText Transfer Protocol
TLS	Transport Layer Security
OPA	Open Policy Agent
LDAP	Lightweight Directory Access Protocol
DAF	Docker Authorization Framework
LTS	Long Term Support
IDE	Integrated Development Environment
Golang	Go Language
URL	Uniform Resource Locator
URI	Uniform Resource Identifier

CHAPTER 1

1.0 Introduction

1.1 Background

Virtualization technology has been widely adopted in the last decade (Bui, 2015). It involves partitioning a computer system into multiple isolated virtual environments. Various virtualization technologies have emerged to the market. These technologies can be classified into two major groups: Hypervisor-based virtualization and Container-based virtualization. Hypervisor virtualization is where each virtual host has a copy of its own Operating system kernel. In container virtualization, all the virtual hosts or containers share the host Operating System kernel. Hence, Container virtualization falls under the Operating System level of Virtualization. This makes container virtualization a more lightweight and efficient form of virtualization that simplify application deployment, portability and configuration compared to hypervisor virtualization. Several container technologies available include LXC, OpenVZ, Linux-Vserver and Docker, with Docker being the most predominant.

When running services, security is always a major concern especially for the service owner. With most services currently being run on virtual environments, we have many security concerns that need to be addressed. Hypervisor-based virtual environments are believed to be more secure than container-based virtual environments since they provide an extra isolation layer between the host and applications (Bui, 2015). An application running within a hypervisor virtual environment can only communicate with the guest Operating System kernel and not the host Operating System kernel. For an application to communicate with host kernel it must first bypass the guest Operating System Kernel then the hypervisor. On the other hand, Containers communicate directly with the host Operating System kernel. This has raised a lot of security concerns, like, if a container or several containers or applications running in them are compromised, then they can be used to attack and compromise other containers or the host system (Chelladhurai, Chelliah, & Kumar, 2016).

Container 1	Container 2
Virtualization Layer i.e. Docker Engine	
Host Operating System	
Hardware	

Figure 1: Container-based Virtualization Architecture (Sultan, Ahmad, & Dimitriou, 2019)

Applications	Applications
Guest Operating System	Guest Operating System
Hypervisor	
Host Operating System	
Hardware	

Figure 2: Hypervisor-based Virtualization Architecture (Sultan, Ahmad, & Dimitriou, 2019).

From Figure1 and Figure 2 above we can see the comparison between container-based virtualization and hypervisor-based virtualization architectures. Figure 1 above shows how the architecture of container virtualization. Where there are multiple containers sharing the same host kernel. Containers communicate to the host through the container engine which is also where the containers run on. Figure 2 above shows that in hypervisor based virtualization, each virtual environment has it s own Operating System kernel. Thus the virtual environments do not communicate with host Operating System kernel. On this paper we are going to focus on Docker containerization technology using Linux kernel. The Docker engine is composed of Docker daemon which is responsible for managing and executing containers running on top of it. And, Docker client, responsible for providing an interface that users use interact with the containers. The user requests and commands are sent to the Docker daemon by the client through RESTful APIs.

With various security concerns in virtual environments, an access control mechanism is needed to prevent illegal or legal entities from illegally accessing unauthorized resources. Currently existing Docker access permission techniques are based on the Linux Kernel autonomic access control mechanism, namespace mechanism

and control groups' mechanism. They provide the basis for access between the containers and the Linux kernel. However certain Docker container processes can gain access to unrelated kernel resources like /sys or /proc in the root directory. This can lead to leakage of kernel resources and cause a Docker container user to gain control permissions to the entire host system (Lang, Jiang, Ding, & Bai, 2019).

Linux access control model is made up of three types namely: Discretionary Access Control, whose flexibility and autonomy allows users to grant some or all their access permissions to another user. This leads to the system's failure to control the flow of information and to determine a specific user permission scope. Secondly, we have the Role-Based Access Control which relies on the idea of adding roles between users and permissions. It controls the grant and revoke user permissions using roles. Finally, the Mandatory Access Control. Docker is based on the Linux default Autonomous Access Control model (Lang, Jiang, Ding, & Bai, 2019).

Docker daemon provides actions and alerts based on policy violations, incident captures and event history. A hardened Docker Container provides various security features like breakout protection and prevention of direct user access by the kernel. Docker has an authorization framework that is not capable of implementing security functions but provides a base for their implementation. The framework works by extending Docker daemon through the REST interface to external authorization plugins. The plugins are responsible for implementing mechanisms for allowing or denying user requests (Hauser, Schmidt, & Menth, 2019.

1.2 Problem Statement

Ideally to effectively achieve adequate security for a system you must first factor in its access control. Access control will help ensure that only legal users can access and perform authorized actions within that system. With high adoption of containers as the ideal way of virtualization, and the ultimate way for deploying micro services. Then, container access control mechanisms need to be addressed. Since without proper container authorization mechanisms there could be serious security breaches.

With the current system we have a problem where any privileged or sudo user in Linux Operating System can access the containers as root with all privileges. Hence, they can modify the contents of any container running within the Linux host system. The

access request does not require to be authorized by the default container configuration since they are privileged users on the host.

This research will review the shortcomings of the existing container access control plugins, evaluate how the container engine is used in the access control process. Since all requests to the containers go through the container engine. And, try to find an effective solution towards container access control by designing and developing a plugin model that will address the current challenges of access control in containers.

1.3 Objectives

i. To review the concept of access control in container virtual computing environments.
ii. To evaluate the existing container authorization frameworks in terms of how they implement access control.
iii. To design and develop an access control plugin model that will make access decisions to containers based on a specific container virtual environment user.
iv. To test and evaluate the performance and effectiveness of the developed access control plugin in container based virtual environments.

1.4 Research Questions

i. How does the container engine allow the authorization plugins to achieve access control in containers?
ii. Can we find an efficient solution to access control in container-based virtual environments?

1.5 Significance

The results of this research will help container virtualization administrators, to be able to control who has access to a specific container and what actions they can perform in the container. The container administrators will be responsible for creating container users, giving them privileges within a specific container and granting them access rights to the container on the policy file.

1.6 Justification

Container-based virtualization is becoming common in many applications and microservice deployment as they reduce the process, development process and management of applications. Containers are offering a lightweight form of virtualization where the applications running on containers in a similar host use a common kernel. This

leads to better resource utilization and improvement in efficiency. However, having all containers applications communicating directly with the host kernel is less secure compared to hypervisor virtualization where each guest Operating system applications communicate with the guest kernel and not that of the host.

The purpose of this research is to identify the shortcomings of existing Docker authorization plugins and determine how we can improve the security of Docker containers by strengthening the access control scheme. This will ensure that any user accessing and executing any action within a Docker container will have to be authorized by the container administrator. This will help in mitigating the current issue where any privileged or sudo user in a Linux host can access and execute actions on Docker containers without having to be authorized. Additionally, we will try to address the issue of default root access to the containers by all users. Users should be allowed access in unprivileged mode then they can elevate their privileges depending on rights given by the administrator.

CHAPTER 2

2.0 Literature Review

Various researches and implementations have been made to improve security in container-based virtualization technology. Container-based virtualization makes the virtualization layer to run as an application running within the Operating System. Since containers interact directly with the host Operating System kernel. We are going to look at and analyze access control mechanisms in container virtualization to identify security vulnerabilities. This research will be focused on Docker containers running in Linux Operating System.

2.1 Containerization

Containerization is a lightweight form of virtualization that consumes less space and time to start. Containerization is a form of virtualization that is similar and different in some ways to a virtual machine. Containers are lightweight compared to virtual machines since they do not encapsulate the entire Operating System and its services. An API is used to make calls for Operating System resources. A hypervisor-based virtual machine includes an entire Operating System and the application, thus for each different virtual machine an entire Operating System must be put in place. In containerization multiple containers can run on a single Operating System sharing similar OS kernel. A container contains the entire runtime environment including: the application, application run time dependencies, libraries, settings, system tools, binaries and configuration files, all bundled together into a single package. Thus, containers provide lightweight application virtualization, isolation of its performance, fast and flexible deployment and fine-grained resource sharing (Pahl, Brogi, Soldani, & Jamshidi, 2017).

2.2 LXC and LXD

These are Linux containers with LXC preceding LXD. These containers run only a single application in their virtual environment. LXC was the origin of container revolution. Since it was the underlying technology for Docker, CoreOS and LXD (Jain, 2016). Linux namespaces are used to limit what processes running in the containers can do on the host system (Kenlon, 2020).

2.3 Docker Containers Overview

Docker containerization is an open source technology capable of building, shipping and running distributed applications. LXC is the underlying technology that is used to implement Docker. Docker is widely used than other container technologies because: Applications packaged in a Docker container can run almost in all Operating systems without requiring any modifications. Secondly with Docker, one can deploy more virtual environments within the same hardware compared to other technologies. Finally, Docker interacts well with third party tools that aid in deployment and management of containers, such as: kubernetes, ansible and Mesos (Bui, 2015). Docker consists of two mainly used components namely, Docker Engine and Docker Hub.

Docker engine consists of Docker daemon which is responsible for execution and management of Docker containers. And, a Docker Client that provides a user interface where users can interact with the containers by sending their commands to the engine. Docker adopts namespaces and control groups Linux features in order to create safe virtual environments for its containers. The namespaces are responsible for isolating users, processes, networks and devices. This is achieved by wrapping the Operating System resources into different instances, giving containers an illusion that they have their own dedicated resources. The Linux namespaces adopted by Docker are; mount, hostname, inter-process communication, network and process identifiers. Cgroups on the other hand are responsible for limiting resource consumption.

Docker consists of a Hub, which is a repository where Docker images are published and can be downloaded by different users. Docker image is a combination of a filesystem and parameters. Which is a series of data layers on top of a base image. A Docker base image is based on a specific operating system distribution. For example, an Ubuntu image or a CentOS image. When building containers, one must pull the desired image from the Docker hub, for example Apache image, then run it to create the container. Thus, a container is a running instance of a specific image. Docker containers run a single application per container. Hence, if you must run another application or another instance of the same application you must run another Docker container (Victor, Marite, & Gundars, 2018).

2.3.1 Docker Security

To ensure that only authorized users access authorized resources securely in containers, the access control technology in use should be flexible, controllable and scalable. Docker is based on autonomous access control technology that is default for Linux kernel and Role access control technology in Selinux. Role access control idea is based on adding roles among users and permissions through roles (Lang, Jiang, Ding, & Bai, 2019). Almost all privileged processes like; ssh and cron are managed by a support system rather than the container. For instance: ssh access is managed by ssh service running on the host. Thus, the kernel only allocates some functions to Docker containers without having to be the real root permissions, in order to satisfy its authority requirements.

Selinux is based on Mandatory Access Control and it separates processes in two ways:

i. Type Enforcement, where a label containing a type is associated with every process and system object. In this process separation, the kernel enforces rules based on permitted actions among the types.

ii. Multicategory Security, where a label assigned to a process or an object can be specialized further with one or more categories, to create different instances of the same type. An access request is accepted only if it is allowed by type enforcement and the process and object are in the same category. Containers are assigned to different categories to ensure that they are separated from each other even if they have a similar type (Bacis, Mutti, Capelli, & Paraboschi, 2015).

Docker access mechanism is also based on Namespaces and Control Groups on the Linux kernel level to achieve isolation. The Namespaces are used to isolate processes, networks and devices. The Control groups are responsible for limiting, measuring and controlling the system resources being used by Docker processes.

2.4 Access Control

Access control is a combination of users' authentication and authorization, access permission operations, license agreement policies and digital rights management (Shoeb & Sobhan, 2010). Authorization is a security function responsible for specifying access privileges to various resources, by ensuring only the resource who is trusted to use a service will have access to it. Authentication is a security function that determines whether a user is valid as who they claim to be. An access control mechanism is a logical component that serves to receive access request from a subject, to decide and enforce the

access decision. Access control mechanisms are deployed to protect objects by mediating requests from subjects.

2.5 Granularity

Granularity means level of detail. In authorization it means the level of details used to put on authorization rules for evaluating a decision to grant or deny access. There are two types of granularity: Coarse, where the authorization for accessing a resource is based on a check for an instance, and the associated roles. And, fine granularity, where access to a resource is based on more details regarding the user or current environment. Attribute Based Access Control (ABAC) is a fine-grained authorization system that defines attributes for access control as: Subject or user, action, resource and environment.

2.5.1 Attribute Based Access Control (ABAC)

This is a logical access control model that controls access to objects by evaluating rules against attributes of entities, where entities are subjects and objects. Logic access control main purpose is to protect objects from unauthorized operations. A subject must satisfy the access control policy established by the object owner, to be authorized to perform the desired operation. Each object in the system must be assigned specific attributes to characterize it, each object must also have at least one policy to define the access rules for the allowable subjects, operations and environment conditions to that object (Hu, et al., 2014).

i. Attributes are characteristics of the object, subject or environment conditions. The information in attributes is defined by name-value pair. Attributes can be categorized into:
    - Subject attributes – used to describe the user attempting access. They may include role, department or job title among others.
    - Action attributes – used to describe the action being attempted. They may include, read execute. Delete among others.
    - Object attributes – used to describe the object or resource being accessed. They may include the object type, the sensitivity or the location among others.
    - Environment attributes – these attributes deal with time, location or dynamic aspects of the access control scenario.

ii. Subject is a human user. Subjects can have one or more attributes.

iii. An Object is a system resource whose access is managed by ABAC system.
iv. An Operation is the execution of a function upon an object at the request of a subject
v. Policy includes the rules and relationships that are used to determine if a requested access should be allowed given the subject and object attributes and possibly environmental conditions.
vi. Environmental conditions are situational context in which access requests occur.

Allowable operation rules are expressed through two forms of computational language: As a Boolean combination of attributes and conditions that satisfy the authorization for an operation. And, as a set of relations associating subject and object attributes and allowable operations. In Boolean language, the rules contain the 'IF' and 'THEN' statements regarding the user making the request, the resource and the action the user wishes to perform.

The ABAC architecture is composed of:

i. Policy Enforcement Point – responsible for protecting objects. It inspects a request and generates authorization request from it and sends to the policy decision point.
ii. Policy Decision Point – Responsible for evaluating incoming requests against the policies configured in it. It returns a permit or deny decision.
iii. Policy Information Point – responsible for bridging policy decision point to external attribute sources, like the LDAP server.
iv. Policy store – collection of logical rules and policies that guide access decisions.
v. Policy Editor – A software tool that allows administrators to create and edit policies that are evaluated and enforced by the decision engine.

Figure 3 below shows how the above five EBAC components interact with each other to perform authorization.

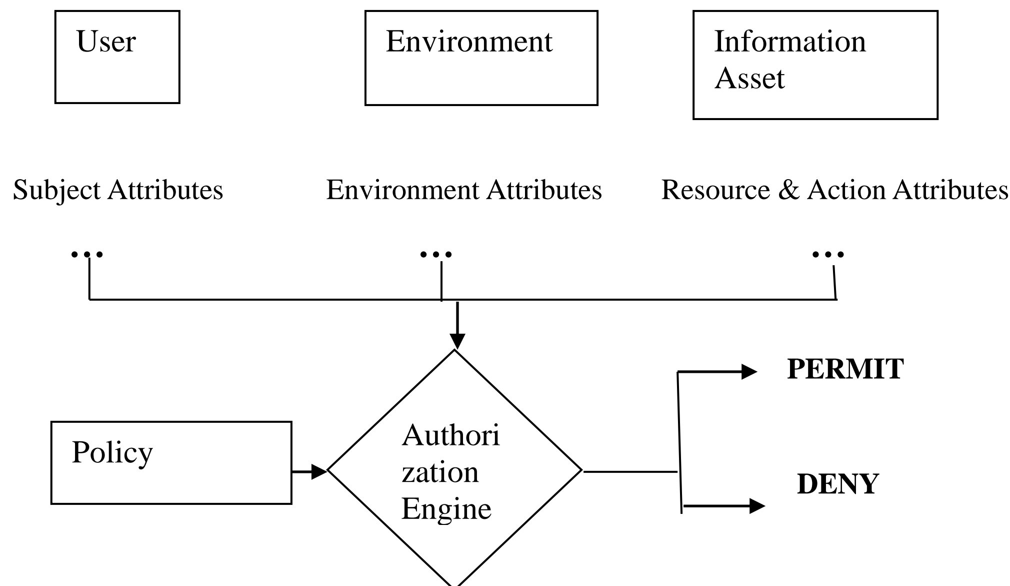


Figure 3: EBAC Methodology (Good, 2018).

2.6 Related Works

2.6.1 AuthZ Plugin

AuthZ plugin works by approving or denying client requests of authentication and commands to the Docker daemon. Different commands can only be run by different users or groups, according to authorization set. When a HTTP request is made to the Docker daemon, the authorization subsystem passes the request to the installed authorization plugin. This request contains the user and the command they wish to run. The plugin is responsible for making the decision whether to allow or deny the request.

User credentials and tokens are not passed to the authorization plugins. Thus, proper authentication and security policies are not enabled on the plugins. To achieve this the authorization plugin needs to be designed in a way that it will provide means that will allow configurations from an administrator *(*Levin, 2016*)*.

Figure 4 below shows the steps involved for an authorization request to become successful in Docker containers. While Figure 5 below shows steps involved for an unauthorized authorization request.

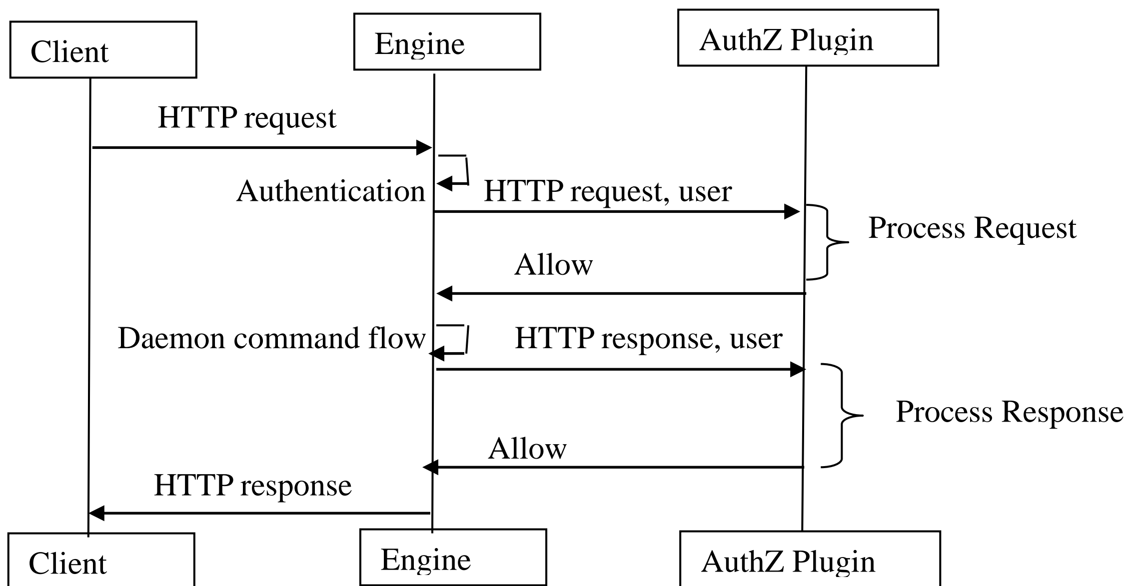


Figure 4: Authorization allow scenario (Levin, 2016).

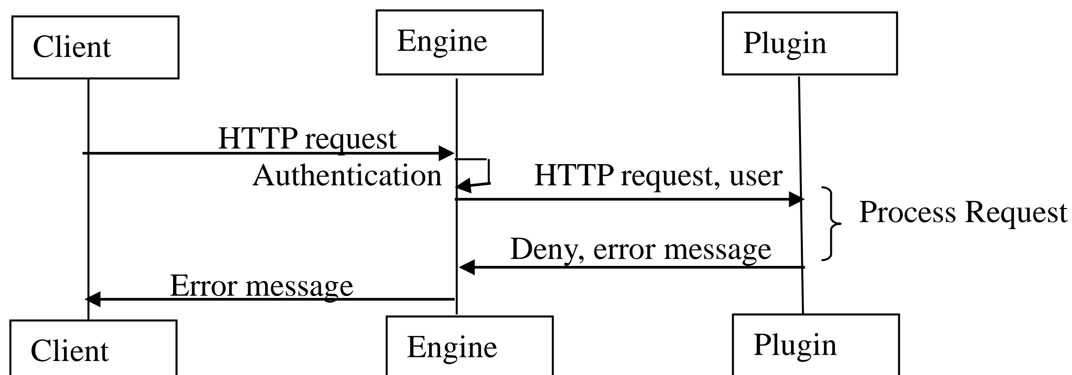


Figure 5: Authorization deny scenario (Levin, 2016)

2.6.2 Distributed Role-Based Access Control (DRBAC)

According to the research by Lang et al titled Research on Docker Role Access Control Mechanism Based on DRBAC, the DRBAC can be used to implement authorization in Docker containers. DRBAC is based on Public Key Infrastructure, for identifying and verifying the identities of entities using a system. It provides a distributed trust management and access control mechanism by describing a controlled behavior based on roles. The roles are defined within a trust domain of an entity and can delegate the role to other roles within different trust domains.

DRBAC allocation syntax is *[Subject → Object] Issuer*; where Object is the role, Issuer is an entity and subject is a role or entity. This means that the signing certificate declares that the role Object grants access to the subject. DRBAC is made of three delegations:

i. Object delegation: [Subject → A.a]A, Publisher A grants A.a role to Subject. A.a role is a definition of A's own namespace.
ii. Assignment delegation: [Subject → A.a]B, B assigns delegate authority of the role A.a to Subject. Afterwards Subject can delegate role A.a to another subject.
iii. Third-party delegation: [Subject → A.a]B, Publisher B delegates role A to Subject, where A and B are different entities.

DRBAC uses PKI identity authentication to verify the legality of a user identity. The permission attribute certificate for a successful user is obtained from LDAP server. Role mapping information and its constraints constitute current user authorization information, which determine permissions given for that role. When a user who has been authenticated to access a system, any access request is authenticated to authentication server on user's rights and a determination is made to determine the subsequent operation sequence.

According to Lang et al, for an administrator to be able to grant roles to different users within Docker, the administrator must have root user and sysadm_t domain. However, if the administrator has root and sysadm_t they can modify the policy source code file directly, so that it can grant all the roles. Thus, the administrator should only have dsm_r role. DRBAC uses proxy delegate, where Docker users enter the access control module to authenticate their identity. They query the LDAP directory server to verify if they are legit. If they are legit, they get a corresponding attribute certificate for user binding, otherwise the user will be denied access to the container. The access control module makes decisions based on current user's permission information and re-defined access control strategy.

2.6.3 Open Policy Agent Docker Authorization Plugin (OPA)

OPA is a policy engine that can be used to implement fine-grained access control for any application. It can also be used in microservices authorization, where it authorizes any request coming to a microservice before it is processed. The decision is made through an API call from the microservice to OPA. OPA uses TLS to allow the Docker daemon to

authenticate the user. The user's X.509 certificate subject common name, must be configured with the user who is the subject of the authorization request. For OPA plugin to work it needs to be made aware of the policy file location. The plugin should be configured with a bind mount; /etc/docker mounted at /opa inside the plugin's container, to provide user-defined OPA policy. OPA uses three inputs to make a policy decision (Nosek, 2019):

i. Data – Which is a set of facts about the outside world. This could be a list of users and their granted permissions.
ii. Query Input – It triggers the computation leading to the decision to be made. Specifies the question in JSON format whose answer will be decided by OPA. For instance; the question, is user Titus allowed to invoke GET /protected /resource? The JSON query will look like: Titus, GET, and /protect/resource.
iii. Policy - It specifies the computational logic, for the given data and query input, yields a policy decision which is a query result. The computational logic is a set of policy rules.

To make a policy decision the three inputs above are fed into the policy engine. The engine interprets the policy rules and makes a policy decision based on data and query input (Nosek, 2019).

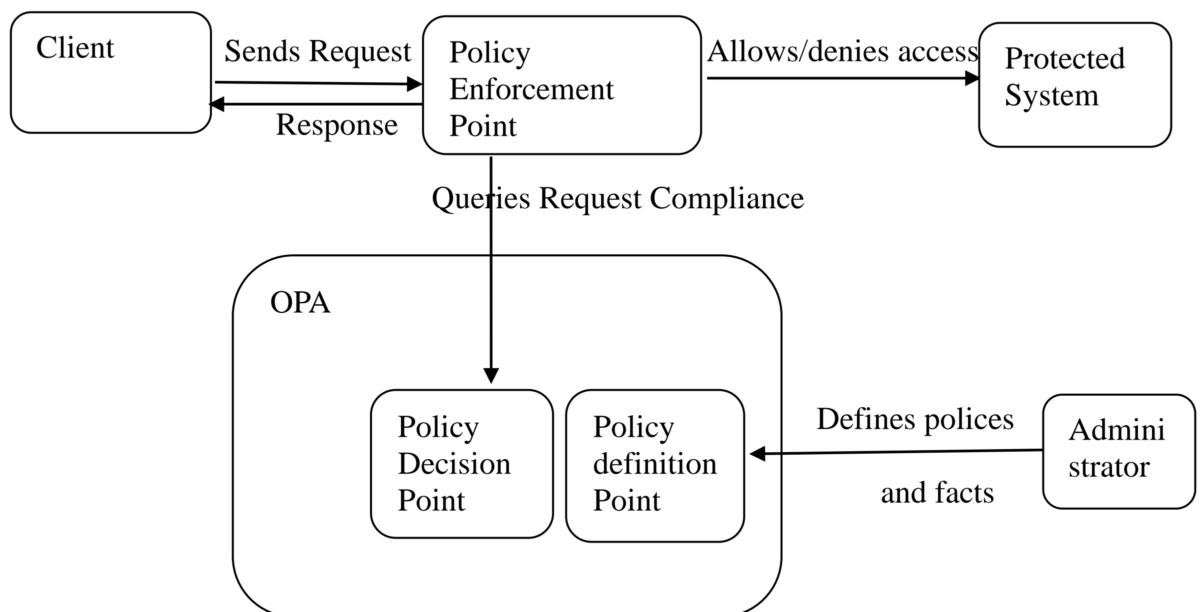


Figure 6: Open Policy Agent Architecture (Block & Spazzoli, 2019)

From Figure 6 above, policy enforcement is not part of OPA. An external agent is used to enforce for the system. OPA is just queried for request compliance and it will respond with allow or deny result, then pass the responsibility of enforcing this decision to the agent.

2.6.4 docker-authz-plugin by Everett Toews

This authorization plugin model was supposed to control access to Docker daemon by approving or denying all requests. The model denies all requests unless one first runs the hello-world image. After running this image, all other requests will be authorized by the plugin to the Docker engine (Toews, 2016).

2.7 Docker Administration

Docker containers are managed locally in Linux kernel using UNIX socket. This enables a user within the Linux system be able to use the CLI to communicate with the Docker engine to execute actions on Docker containers. The Docker CLI is limited only to root users and Docker group users. Docker employs an all or nothing approach where you either have admin access or no access. Docker does not offer admin segregation controls, where different users can have different admin rights to different containers (Kuusik, 2015).

AuthZ plugin is design accepts user requests from the DAF. However, it does not transfer user information to the plugin leading to lack of proper access control. DRBAC is based on role-based access control which is coarse-grained. DRBAC also allows a subject to assign their roles to another subject. This research will borrow from OPA since it is based on fine-grained access rules and has a decision point in its architecture. However, our architecture will not use an agent for enforcing the decisions like in OPA, rather we will use the inbuilt Docker Authorization Framework for executing the decisions made by the plugin. Also, the policies and attributes will be defined on the Docker container by the Docker administrator rather than in OPA where the Linux system administrator defines them on the plugin. The research also borrows from the AuthZ architecture, on the concept of using the DAF to execute decisions from the plugin.

2.8 Conceptual Model

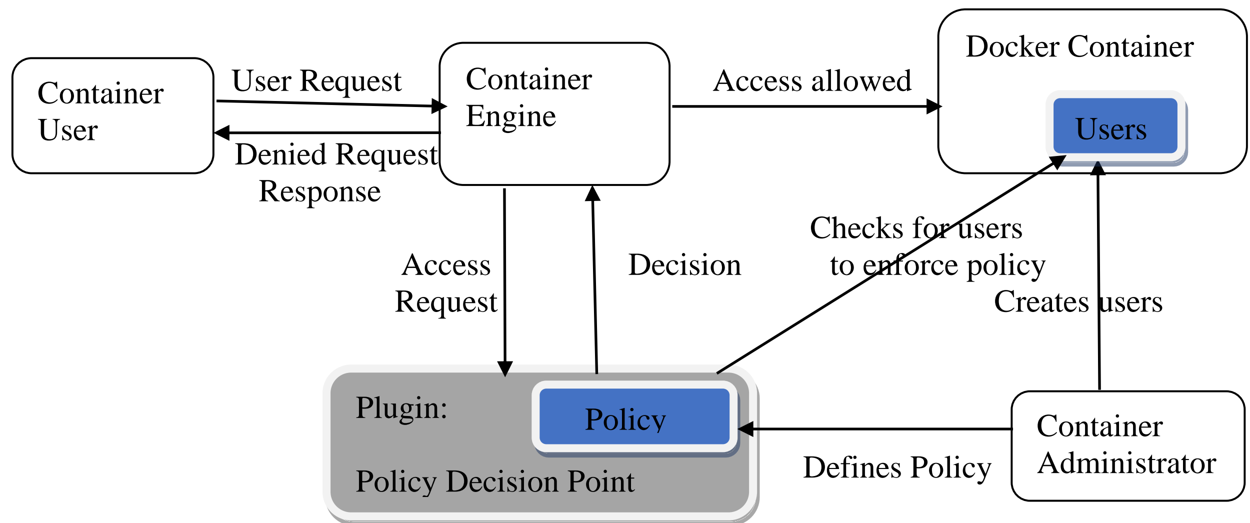


Figure 7: Conceptual Model

From figure 7 above, the container administrator creates users in the container and assigns privileges to them. Some can have super user rights while others cannot. The administrator also defines the policy in terms of which container users can gain access to the container. The policy is stored within the host machine where permissions have been restricted to the administrators group only. When a container user sends an access request to the Docker engine. The request is forwarded to the access control plugin which decides to allow or deny the request based on the defined policy. On the policy the administrator defines users who are allowed access to the container. Thus, the plugin checks the user in the request URI against user defined in the policy and users in the container. If there is a match, by the requesting user is allowed access in the policy and is a user in the container they wish to access then the request is allowed. Otherwise, the request is denied.

CHAPTER 3

3.0 Research Methodology

This research will be based on exploratory research design. Since it is main aim is to understand container access control mechanisms by exploring the current problems from default container access control mechanism and that adopted by other access control plugins. And, propose a possible efficient solution to the problem.

3.1 Research Design

This research will review the container engine and current container access control plugs in use. The main aim being to find an efficient way of solving access control in container virtual environments. Meaning, the result of this research will lead to development of a more efficient access control plugin for containers compared to the current access controls in place. It will include selection of a population that is composed of container virtualization experts within Nairobi, who will help determine key themes of access control in containers. A sample of the experts will be selected and involved in the data gathering and usability testing phases. This plugin development will use the waterfall software development method of software development. The flow will be as in figure 8 below.

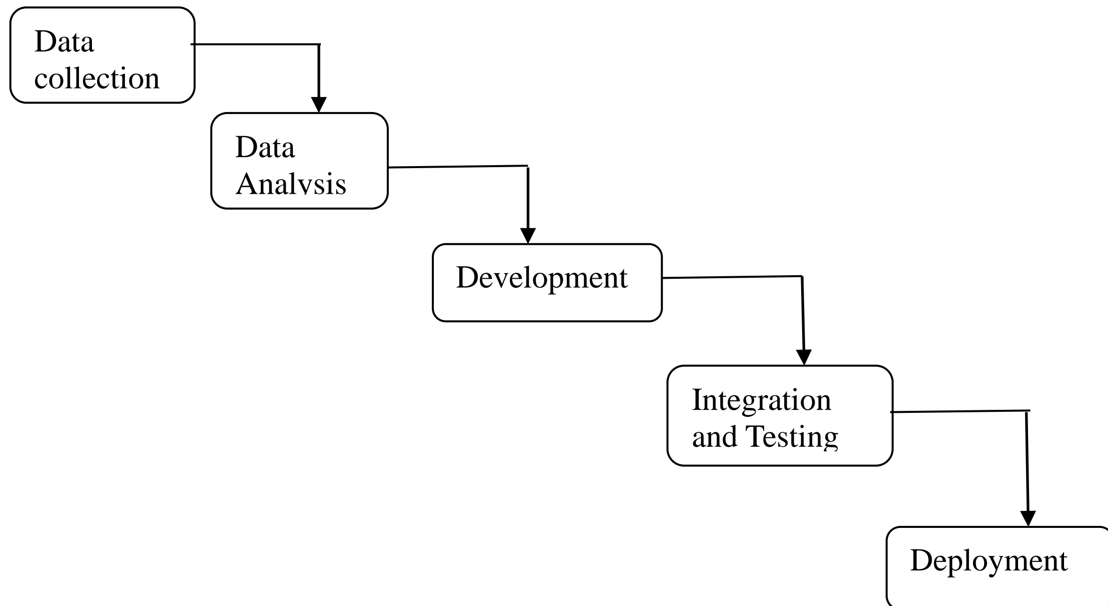


Figure 8: Research Design Method

3.2 Population

This research will be conducted by including inputs from container virtualization experts within Nairobi, Kenya.

3.3 Sample Size

The sample will target ten Container virtualization experts within Nairobi. The sample selection will be based on snowballing technique where, three known Docker experts, two from DTOne and one from Angani cloud services will be engaged in the research, then asked for their referrals to other container virtualization experts within Nairobi. DTOne and Angani are selected as the first sample because they use container virtualization environments for running their applications. And, there is a great possibility of accessing and interacting with their container experts. The selected sample will be involved in the interviews and focus groups. Three of them will also be involved in the tests that will be conducted during the research.

3.4 Data Collection

The data to be collected will be qualitative. It will be collected through.

3.4.1 Focus Groups

The discussion will involve the ten container virtualization experts from Nairobi region. The main aim for the focus group will be to determine:

i. The most widely adopted container virtualization technology in Kenya.
ii. The access controls used by containers to prevent unauthorized access.
iii. How comfortable are administrators with the container default access control model? If not, how would they welcome an access control plugin that restricts users from accessing the containers based on a set policy.
iv. The most ideal Operating System where containers can be deployed to perform optimally.

3.4.2 Observations

This will involve observing the tests that will be conducted on Docker authorization framework and existing plugins. The tests will involve:

i. How the container engine accepts requests. This includes the format and syntax.
ii. How current container access control plugins have been implemented. In terms of how they control access to containers.
iii. To test how the container engine interacts with the plugins. How the forwarding of requests and responses happen between the engine and the plugin.
iv. Observing how the policy should be defined to be understood by the plugin and be analyzed to decide if a user request is valid or not.

These tests should help find the existing gaps that have not been identified by current container access control plugins. And, try to find an efficient solution to address these gaps. The tests will also provide a guideline on how to develop a plugin that will communicate with the container engine efficiently.

3.5 Data Analysis

The data collected from focus group will be analyzed to understand and summarize the main features and themes between the current container technologies. It will address why some container technologies are more adopted than others. And, determine the pros, cons and features of default and current access control mechanisms in container

technology. To get this information. The data will be analyzed using content analysis. This involves finding similar concepts on access control from all container virtualization experts. The similar concepts will build a stronger ground regarding what is missing together with what experts what in an ideal container access control framework.

Data collected through observations on tests conducted on existing container authorization plugins will be analyzed using framework analysis. Where, related themes, patterns and concepts will be mapped to identify the key issues that need to be addressed in container access control. Also, the analysis is supposed to determine the ideal way of communicating with the container engine, to help find an efficient solution to container access control gaps.

3.6 Plugin Development

Once the data has been analyzed, and require patterns, themes and gaps identified then a plugin model will be developed to address the gaps identified. The plugin will be developed using python programming language. The development will involve developing individual system components and ensuring that they can communicate with each other as expected. Also, the plugin will be developed to achieve efficient communication with the container engine. Thus, we will be evaluating how plugins communicate with the container daemon to ensure we develop a usable product.

3.7 Testing the Plugin

The developed access control plugin will be tested through.

i. Unit testing - to ensure that the various components that will be developed or whose parameters will require to be set, are working perfectly as expected on each component.
ii. System testing – to ensure that all arising access control requirements have been addressed by the final product. And, that the product is interacting well with its intended environment without affecting container performance in Linux Operating System.

iii. Usability testing - will be done to ensure that the system can be deployed and used easily by small scale container administrators running Docker containers in Linux environments.

3.8 Tools

The tools to be used in this research include:

i. A computer running Linux Ubuntu Operating System Ubuntu 18.04 LTS.
ii. Docker Engine version 19.03.8
iii. PyCharm-community IDE.
iv. Postman – API testing tool.
v. Docker-Machine version 0.16.0.
vi. Microsoft Excel for matching themes and patterns in data collected.

CHAPTER 4

4.0 Results and Discussions

We are going to review the results obtained from the research design and discuss them against expected outcome.

4.1 Results

4.1.1 Selected Sample Size

The sample for this research included five container virtualization experts within Nairobi. The selection technique was based on snowballing. Where three container virtualization experts that were well know were asked if they knew other experts in that field. The first three experts were from DTOne organization. They referred four more experts, who referred two more experts. Nine container virtualization experts were consulted but only five were around and willing to join a focus group and discuss the container access control issue.

4.1.2 Focus group data

A single focus group discussion involving five container virtualization experts in Nairobi was held on first day of February 2020. The experts were from the sample of experts gathered during the snowballing sampling. Only the willing and available experts were involved in this discussion. The focus group discussion was based on the subject matter of access control in container based virtual environments. The discussion points included:

Most widely used containerization technology and the reason – the discussion concluded that the most widely adopted container virtualization technology is Docker. Docker has been widely adopted because it is easier that other approaches. Also, docker has partnered with google and Red Hat and, docker containers can be deployed on any Operating system.

How access control has been implemented in container virtual environments? It was noted that the current container based virtual environments like Docker and LXC do not have an inbuilt authorization technique. All users in a system can access the containers in root mode. Thus, can edit the contents or applications running from within the container.

The best Operating system to test a container access control plugin on. Since container technology is built on Linux kernel and most servers that run containers in various organizations and cloud environments are Linux based, it was agreed it would be feasible to use a Linux Operating system. This would ensure efficient testing without having to involve a lot of dependency issues. If an access control plugin works on a Linux kernel, then it would be easy to deploy the it in other operating systems. Since they run within inbuilt Linux-based environment even on other Operating Systems. Like for Docker, a Docker machine must be installed on other Operating Systems. Inside the docker machine the command line is based on bash. This explained why most of the servers used to run the containers are Linux based.

What would be an ideal access control plugin for container virtualization? It should first and most importantly limit the users who can access the container virtual environment. Access should be based on a user already within the container to limit the root access for containers by everyone. Then extend access control to users on the host machine if possible.

4.1.3 Testing and Observations

This was conducted by testing how the container engines handle user requests. This is in terms of the key standards in terms of; syntax that the framework uses to identify requests from users and its response syntax. It was determined that the basic principle of docker plugin infrastructure is that it must work from certain defined directories. The plugin uses HTTP requests to communicate with the container engine. Plugs are discovered by being searched on the designated plugin directories. There are only three types of files that can be put in a plugin directory (docker Inc, 2019).

i. .json – files containing full json specification for the plugin.
ii. .sock – UNIX domain sockets.

iii. .spec – files containing URLs, like tcp://localhost:port_number

It was observed that the syntax for an access control request in container environments can only include: User, Request URI, Request Method, Request Body and Request header. And, the response allowed is of the syntax: Allow, Message and Error if any (docker Inc, 2019).

Also, tests were done on the existing plugins to determine how they achieve access control in Docker virtual environments. This was aimed at identifying the gaps and issues that they have not addressed. Two current access control plugins for Docker containers were put into test. Open Policy agent (OPA) for Docker authorization and authz by Everett Toews. The plugins were tested in an Ubuntu based Linux kernel. The authz plugin was tested within a Docker-Machine, where a virtual environment was created, and dependencies installed. OPA was tested on a Docker engine running on Ubuntu 18.04 LTS. Access control tests were done on both containers following the documentation and directions given by the developers.

From the tests on the two plugs we were able to observe that authz does not use any users to achieve access control to containers. Rather it denies all docker commands to execute unless you first pull and install the hello-world image from Docker hub. After running this image everyone on the host machine can run all docker commands even access the containers. OPA does not depend on host or container users. Rather it uses users and permissions defined on the policy. Each user is linked to a read-only permission either with true al false rights to do so. To test access control a user from the policy is loaded in json format to a configuration file located on ~/.docker directory. If the user has read-only access then they cannot execute write commands, but if the user loaded to json configuration file is allowed all rights then they can run all commands even access the containers without being exempted.

4.1.4 Analysis of collected data

The data collected from the focus group was qualitative in nature. It was analyzed to understand and summarize the main features of container based virtual environments and how access control can be implemented. Content analysis method was used to analyze this data to find common features and main themes from information that was discussed

by the container virtualization experts. The concepts gathered from collected data helped to determine that Docker is the most widely used container technology. This is because it was the main key word that came to the experts when they heard of container virtualization. All of them had interacted with Docker technology and only one who had interacted with another technology. A general pattern that was noted was that Docker administrators have a challenge of controlling container access since, any host machine user with super user rights can access any container as root the root user, default in the containers. Leading there to be no need for creating specific container users. This posed as a big challenge in auditing containers since you must rely on host system logs to make major decisions like determining which user accessed a container at a given time. Refer to table 1 below to find common themes among the container virtualization experts.

Code	Description
Most adopted Container virtualization Technology	Docker since it is easy to use and can be used with all operating systems
Widely adopted Operating system for use with Docker	Linux Kernel since it is built from on LXC. Also, most servers run on Linux environments
Is access control in containers an issue?	yes, since anyone with super user access to host system can access the containers as root. By running the exec command.
Ideal Container access control plugin	Should make access decision based on users already in the system. Container users and if possible, host system users also

Table 1: Content Analysis

The data collected from performed tests was also qualitative in nature. It was analyzed using framework analysis. The key patterns and themes used in the current container authorization plugins was determined. Also, the syntax of the container engine authorization framework was analyzed to determine how the plugins are written to be able to communicate with the framework. The analysis is as per table 2 below. It was found that since Docker and LXC are developed using Go language then some of the plugins like OPA are designed using the same language for easy integration. The authz plugin was developed in python to simplify the code, since Go language is procedural (Toews, 2016).

Code	Description
How docker authorization framework interacts with user requests	By use of HTTP requests and responses
Docker daemon request and response syntax	For an authorization request it gets the following key parts: (User, Request Method, URI, Request body and Request header) For Response it takes a Boolean (True or False) to allow or deny a request, a message and an error if need be.
Languages that have been used to try to develop container authorization plugins	Golang since docker is developed using Golang and python has been tested too using flask framework.
Currently Developed plugins	Open Policy Agent (OPA) using Golang and authz by Everett Toews using python.
How do existing container plugins communicate with the container engine	Using http requests. The code structure follows that of the syntax required by the container engine.
Basic working of the current container authorization plugins	They do perform authorization, but not based on an existing container or system user. OPA defines sample users and permissions in the policy. Authz denies all docker requests unless one runs the hello-world image, then it allows all the authorizations.

Table 2: Framework Analysis

4.1.5 Plugin Development

The container authorization plugin prototype has been developed using Python3 language, based on Flask framework. PyCharm was used as the IDE for developing the prototype. Three main routes '/Plugin.Activate', '/AuthZPlugin.AuthZReq', '/AuthZPlugin.AuthZRes' that are required to implement a Docker authorization plugin were the first to be developed. The regular expression module was used to make container access decisions based on a specific container user on the request URI. The prototype checks for exec command and the username from the request URI.

```python
@plug.route("/info/<query>", methods=["GET"])
def state(query):
    if query=="state":
        qu=(1 if enabled else 0)
        return str(qu)
    else:
        return "-1"

@plug.route("/Plugin.Activate", methods=["POST"])
def start():
    return jsonify({"Implements": ["authz"]})

@plug.route("/AuthZPlugin.AuthZReq", methods=["POST"])
def req():
    plugin_request=json.loads(request.data)
    print(plugin_request)
    response={"Allow":True}
    if search(r'/(exec)$', plugin_request["RequestUri"]) != None:
        docker_request=json.loads(base64.b64decode(plugin_request["RequestBody"]))
        if match(r'^titus$|^elly$', docker_request["User"])!=None:
            response={"Allow":True}
        else:
            response={"Allow":False, "Msg":"You are not authorized to Run Execute command"}
    if not enabled:
        response={"Allow":True}
    return jsonify(**response)

@plug.route("/AuthZPlugin.AuthZRes", methods=["POST"])
def res():
    response={"Allow":True}
    return jsonify(**response)
```

Figure 9: Plugin major components Development

From figure 9 above we can see the three main routes required for a plugin to interact with the container engine. They are all implemented using HTTP POST method. The '/info/state' route uses GET method. It shows if a plugin is running or not. The '/AuthZPlugin.AuthZReq' route is the main policy that is used to make authorization decisions. It makes decisions based on exec command and username provided on the request URI. The '/Plugin.Activate' is used to test that the plugin works by displaying specified information. The '/AuthZPlugin.AuthZRes' returns the plugin responses based on decision.

The file used to discover the plugin on the plugin directory by the container engine is '.spec, using tcp://localhost:6000' as shown on figure 10 below.

```
docker-authz
/home/trugendo/PycharmProjects/auth_test/venv/bin/python /home/trugendo/PycharmProjects/docker-authz/docker-authz.py
error: [Errno 2] No such file or directory: '/etc/docker-authz/authz.json'
Error occurred while writing pid file!!
[Errno 13] Permission denied: '/var/run/docker-authz.pid'
 * Serving Flask app "docker-authz" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:6000/ (Press CTRL+C to quit)
```

Figure 10: How Plugin is Discovered by Container Engine

4.1.6 Plugin Testing

The authorization plugin prototype was tested using below ways.

4.1.6.1 Unit testing

Each plugin component functionality was tested using the Postman API testing tool. Individual URL routes ware tested to determine if they function properly as expected. POST and GET requests were sent using the Postman tool to determine if they returned the expected result.

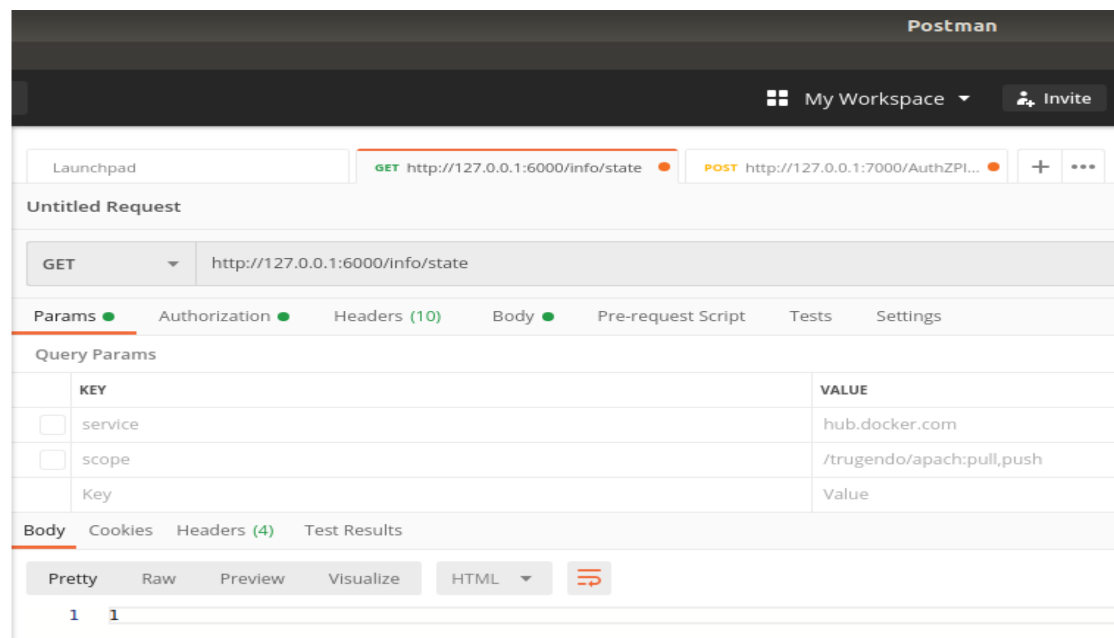


Figure 11: Plugin Information State Module Test

From figure 11 above a test was conducted on '/info/state' route module to determine the state of the plugin while it was running. The result is 1 showing that the plugin is working perfectly.

4.1.6.2 System testing

After finalizing the unit testing and ensuring all URLs and HTTP methods were functioning as required. An Installation bash script was written to deploy the plugin in a Linux Operating System running on Ubuntu Kernel. Docker engine was deployed on the Ubuntu kernel, then Nginx and PostgreSQL docker images were used to create containers

and do the tests. The access control mechanism of the plugin was tested under different scenarios as below.

During the start of testing the policy was defined to allow only one user access to the containers. The user should be already created in the containers that are to be accessed using the specified username.

```
    if search(r'/(exec)$', plugin_request["RequestUri"]) != None:
        docker_request=json.loads(base64.b64decode(plugin_request["RequestBody"]))
        if match(r'^titus$', docker_request["User"])!=None:
            response={"Allow":True}
        else:
            response={"Allow":False, "Msg":"You are not authorized to Run Execute command"}
    if not enabled:
        response={"Allow":True}
    return jsonify(**response)

@plug.route("/AuthZPlugin.AuthZRes", methods=["POST"])
def res():
    response={"Allow":True}
    return jsonify(**response)
```

Figure 12: Initial Policy definition

From figure 12 above the policy is defined to allow only user ‘titus’ access to the containers using the exec command. Both the username and exec command should be included in the URI for the access to be allowed.

Access control for host system users

Normally all host users with privileged rights can access the containers in root mode. According to our policy all host system users are restricted access to the containers. No user from the Host system should be able to access any container when the plugin is working.

```
root@titus:~# docker ps
CONTAINER ID    IMAGE          COMMAND                  CREATED       STATUS         PORTS                NAMES
095bf6b1f4ff    e3faa30ac980   "docker-entrypoint.s…"   3 days ago    Up 2 minutes   5432/tcp             postgres_authorization_test
81a601d9d09a    c3502de97de8   "nginx -g 'daemon of…"   3 days ago    Up 2 minutes   0.0.0.0:80->80/tcp   nginx_authorization_test
root@titus:~# docker exec -it nginx_authorization_test bash
Error response from daemon: authorization denied by plugin docker-authz: You are not authorized to Run Execute command
root@titus:~# docker exec -it postgres_authorization_test bash
Error response from daemon: authorization denied by plugin docker-authz: You are not authorized to Run Execute command
root@titus:~#
root@titus:~# docker exec -it -u titus nginx_authorization_test bash
titus@81a601d9d09a:/$ ls /home
rugendo  titus
titus@81a601d9d09a:/$ exit
exit
root@titus:~# ls /home
elly  pascal  titus  trugendo
root@titus:~# docker exec -it -u pascal nginx_authorization_test bash
Error response from daemon: authorization denied by plugin docker-authz: You are not authorized to Run Execute command
root@titus:~# docker exec -it -u elly nginx_authorization_test bash
Error response from daemon: authorization denied by plugin docker-authz: You are not authorized to Run Execute command
root@titus:~#
root@titus:~#  docker exec -it -u titus postgres_authorization_test bash
titus@095bf6b1f4ff:/$ ls /home
elly  rugendo  titus
titus@095bf6b1f4ff:/$ exit
exit
root@titus:~# docker exec -it -u trugendo postgres_authorization_test bash
Error response from daemon: authorization denied by plugin docker-authz: You are not authorized to Run Execute command
root@titus:~#
```

Figure 13: Host users access test

From figure 13 above, all host users have been denied access to the plugin when they try to run the exec command. Also, if you specify a host user who is not in the container, we can see that the request is denied by the plugin.

Access control for container users

According to defined policy on figure 12 above only container user 'titus' is allowed access to the containers. Any other container user should not be allowed by the plugin to access the containers even if they are already created in the container.

```
root@titus:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                NAMES
095bf6b1f4ff        e3faa30ac980        "docker-entrypoint.s…"   3 days ago          Up 11 minutes       5432/tcp             postgres_authorization_test
81a601d9d09a        c3502de97de8        "nginx -g 'daemon of…"   3 days ago          Up 10 minutes       0.0.0.0:80->80/tcp   nginx_authorization_test
root@titus:~# docker exec -it -u titus postgres_authorization_test bash
titus@095bf6b1f4ff:/$ ls /home
elly  rugendo  titus
titus@095bf6b1f4ff:/$ exit
exit
root@titus:~# docker exec -it -u elly postgres_authorization_test bash
Error response from daemon: authorization denied by plugin docker-authz: You are not authorized to Run Execute command
root@titus:~# docker exec -it -u rugendo postgres_authorization_test bash
Error response from daemon: authorization denied by plugin docker-authz: You are not authorized to Run Execute command
root@titus:~#
```

Figure 14: Container Users access Test

From figure 14 above we can see that only user ‘titus’ can access the container. Users ‘elly’ and ‘rugendo’ are created within the container but since they are not allowed access by the policy, their access requests get denied by the access control plugin.

Container Users Permission Test

The container administrator creates users and assigns different privileges to them. Based on above tests we have seen that to access a container the URI must contain the exec command and username. But the username is not being authenticated by password as the access is allowed. For security Docker ensures that any user created in the container only gains access in a non-privileged mode then they can elevate to superuser by confirming their password on the inside. Any container user needs to be in root mode or superuser mode to be allowed file write and modification rights within the container. Otherwise, the write or modification requests will be denied.

```
root@titus:~# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED      STATUS          PORTS                NAMES
095bf6b1f4ff   e3faa30ac980   "docker-entrypoint.s…"   3 days ago   Up 25 minutes   5432/tcp             postgres_authorization_test
81a601d9d09a   c3502de97de8   "nginx -g 'daemon of…"   3 days ago   Up 24 minutes   0.0.0.0:80->80/tcp   nginx_authorization_test
root@titus:~# docker exec -it -u titus nginx_authorization_test bash
titus@81a601d9d09a:/$ ls /home
rugendo  titus
titus@81a601d9d09a:/$ id
uid=1000(titus) gid=1000(titus) groups=1000(titus),27(sudo)
titus@81a601d9d09a:/$ id rugendo
uid=1001(rugendo) gid=1001(rugendo) groups=1001(rugendo)
titus@81a601d9d09a:/$ nano /etc/titus_test.txt
```

Figure 15: Container Users Privileges

From figure 15 above we can see that only user 'titus' can elevate the privileges to root mode since he is a member of sudo group. User 'rugendo' cannot be able to modify or write to any files. User 'titus' can modify and write files but only when he has elevated root mode.

```
root@titus:~# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED      STATUS          PORTS                NAMES
095bf6b1f4ff   e3faa30ac980   "docker-entrypoint.s…"   3 days ago   Up 25 minutes   5432/tcp             postgres_authorization_test
81a601d9d09a   c3502de97de8   "nginx -g 'daemon of…"   3 days ago   Up 24 minutes   0.0.0.0:80->80/tcp   nginx_authorization_test
root@titus:~# docker exec -it -u titus nginx_authorization_test bash
titus@81a601d9d09a:/$ ls /home
rugendo  titus
titus@81a601d9d09a:/$ id
uid=1000(titus) gid=1000(titus) groups=1000(titus),27(sudo)
titus@81a601d9d09a:/$ id rugendo
uid=1001(rugendo) gid=1001(rugendo) groups=1001(rugendo)
titus@81a601d9d09a:/$ nano /etc/titus_test.txt
```

Figure 16: User titus creating a file in non-root mode

From figure 16 above user 'titus' tries to create and write file titus_test.txt in non-privileged mode.

Figure 17: User titus request denied

From Figure 17 above we can see that the request to create file titus_test.txt by user ‘titus’ is denied since he is in unprivileged mode.

Figure 18: Unprivileged user rugendo request denied

From figure 18 we can see that user ‘rugendo’ too is denied the request of creating a file called rugendo_test.txt. This is because is an unprivileged user.

```
titus@81a601d9d09a:/$ id
uid=1000(titus) gid=1000(titus) groups=1000(titus),27(sudo)
titus@81a601d9d09a:/$ id rugendo
uid=1001(rugendo) gid=1001(rugendo) groups=1001(rugendo)
titus@81a601d9d09a:/$ nano /etc/titus_test.txt
titus@81a601d9d09a:/$ sudo bash
[sudo] password for titus:
titus@81a601d9d09a:/$ ^C
titus@81a601d9d09a:/$ ls /etc
adduser.conf          debconf.conf    fonts     gshadow-   issue        libaudit.conf  motd           os-release  profile.d  rc5.d        security  subgid    systemd        xattr.conf
alternatives          debian_version  fstab     host.conf  issue.net    localtime      mtab           pam.conf    rc0.d      rc6.d        selinux   subgid-   terminfo
apt                   default         gai.conf  hostname   kernel       login.defs     nanorc         pam.d       rc1.d      rcS.d        shadow    subuid    timezone
bash.bashrc           deluser.conf    group     hosts      ld.so.cache  logrotate.d    nginx          passwd      rc2.d      resolv.conf  shadow-   subuid-   ucf.conf
bindresvport.blacklist dpkg           group-    init.d     ld.so.conf   machine-id     nsswitch.conf  passwd-     rc3.d      rmt          shells    sudoers   update-motd.d
cron.daily            environment     gshadow   inputrc    ld.so.conf.d mke2fs.conf    opt            profile     rc4.d      securetty    skel      sudoers.d vim
titus@81a601d9d09a:/$ sudi bash
bash: sudi: command not found
titus@81a601d9d09a:/$ sudo bash
[sudo] password for titus:
root@81a601d9d09a:/# nano /etc/tito2_test.txt
root@81a601d9d09a:/# cat /etc/tito2_test.txt
file for testing purpose
root@81a601d9d09a:/# ls /etc
adduser.conf          debconf.conf    fonts     gshadow-   issue        libaudit.conf  motd           os-release  profile.d  rc5.d        security  subgid    systemd        vim
alternatives          debian_version  fstab     host.conf  issue.net    localtime      mtab           pam.conf    rc0.d      rc6.d        selinux   subgid-   terminfo       xattr.conf
apt                   default         gai.conf  hostname   kernel       login.defs     nanorc         pam.d       rc1.d      rcS.d        shadow    subuid    timezone
bash.bashrc           deluser.conf    group     hosts      ld.so.cache  logrotate.d    nginx          passwd      rc2.d      resolv.conf  shadow-   subuid-   tito2_test.txt
bindresvport.blacklist dpkg           group-    init.d     ld.so.conf   machine-id     nsswitch.conf  passwd-     rc3.d      rmt          shells    sudoers   ucf.conf
cron.daily            environment     gshadow   inputrc    ld.so.conf.d mke2fs.conf    opt            profile     rc4.d      securetty    skel      sudoers.d update-motd.d
root@81a601d9d09a:/# su - rugendo
rugendo@81a601d9d09a:~$ sudo bash

We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.

[sudo] password for rugendo:
rugendo is not in the sudoers file.  This incident will be reported.
rugendo@81a601d9d09a:~$ nano /etc/rugendo_test
rugendo@81a601d9d09a:~$
rugendo@81a601d9d09a:~$ ls /etc
adduser.conf          debconf.conf    fonts     gshadow-   issue        libaudit.conf  motd           os-release  profile.d  rc5.d        security  subgid    systemd        vim
alternatives          debian_version  fstab     host.conf  issue.net    localtime      mtab           pam.conf    rc0.d      rc6.d        selinux   subgid-   terminfo       xattr.conf
apt                   default         gai.conf  hostname   kernel       login.defs     nanorc         pam.d       rc1.d      rcS.d        shadow    subuid    timezone
bash.bashrc           deluser.conf    group     hosts      ld.so.cache  logrotate.d    nginx          passwd      rc2.d      resolv.conf  shadow-   subuid-   tito2_test.txt
bindresvport.blacklist dpkg           group-    init.d     ld.so.conf   machine-id     nsswitch.conf  passwd-     rc3.d      rmt          shells    sudoers   ucf.conf
cron.daily            environment     gshadow   inputrc    ld.so.conf.d mke2fs.conf    opt            profile     rc4.d      securetty    skel      sudoers.d update-motd.d
rugendo@81a601d9d09a:~$
```

Figure 19:File creation and editing in root mode

From figure 19 above user 'titus' can create and write a file called tito2_test.txt when in privileged mode. User 'rugendo' request to be elevated to root mode is also rejected since he is not a member of sudo group.

Altering Policy Test

The policy file is stored on the host Operating system. For our case it is stored on the Ubuntu 18.04 LTS at '/usr/local/bin/docker-authz.py' absolute path. To protect the policy from authorized access and editing from host system users we created a password protected group called authz and gave it all system privileges. The policy file ownership was changed from default root to main container administrator who for our case was user 'titus'. The group ownership was also changed from default root group to authz group. Read, write and execute permissions were given to authz group members only. All other users were not given any permissions to the file. Also, the default sudo and admin groups were disabled from '/etc/sudoers' file.

```
root@titus:~/docker-authz# cd /usr/local/bin/
root@titus:/usr/local/bin# ls
docker-authz  docker-authz.py  flask
root@titus:/usr/local/bin# ls -lrth
total 16K
-rwxr-xr-x 1 root root  212 Cam 11 19:16 flask
-rwx------ 1 root root 2.1K Cam 16 13:31 docker-authz.py
-rwx------ 1 root root 7.1K Cam 16 13:31 docker-authz
root@titus:/usr/local/bin# chown titus docker-authz.py
root@titus:/usr/local/bin# chown titus docker-authz
root@titus:/usr/local/bin# chgrp authz docker-authz.py
root@titus:/usr/local/bin# chgrp authz docker-authz
root@titus:/usr/local/bin# ls -lrth
total 16K
-rwxr-xr-x 1 root   root    212 Cam 11 19:16 flask
-rwx------ 1 titus authz 2.1K Cam 16 13:31 docker-authz.py
-rwx------ 1 titus authz 7.1K Cam 16 13:31 docker-authz
root@titus:/usr/local/bin# chmod 770 docker-authz.py docker-authz
root@titus:/usr/local/bin# ls -lrth
total 16K
-rwxr-xr-x 1 root   root    212 Cam 11 19:16 flask
-rwxrwx--- 1 titus authz 2.1K Cam 16 13:31 docker-authz.py
-rwxrwx--- 1 titus authz 7.1K Cam 16 13:31 docker-authz
```

Figure 20: Changing policy file ownership

From figure 20 above, the policy file ownership is changed from root to user 'titus' and group ownership changed to 'authz' group. The policy file read, write and execute permissions are assigned to the owner and owner group. No rights are assigned to other host system users.

```
#
# This file MUST be edited with the 'visudo' command as root.
#
# Please consider adding local content in /etc/sudoers.d/ instead of
# directly modifying this file.
#
# See the man page for details on how to write a sudoers file.
#
Defaults        env_reset
Defaults        mail_badpass
Defaults        secure_path="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin"

# Host alias specification

# User alias specification

# Cmnd alias specification

# User privilege specification
root    ALL=(ALL:ALL) ALL

# Members of the admin group may gain root privileges
#%admin ALL=(ALL) ALL
%authz ALL=(ALL) ALL
# Allow members of group sudo to execute any command
#%sudo  ALL=(ALL:ALL) ALL

# See sudoers(5) for more information on "#include" directives:

#includedir /etc/sudoers.d
root@titus:/usr/local/bin#
root@titus:/usr/local/bin#
```

Figure 21: Denying all other groups all privileges

From figure 21 above, all default groups that have privileged access in Linux kernel have been disabled. The group ‘authz’ has been given all privileges. This will allow only members from this group to access and modify the policy file.

```
root@titus:~/docker-authz# ls /home/
elly  pascal  titus  trugendo
root@titus:~/docker-authz# id elly
uid=1002(elly) gid=1003(elly) groups=1003(elly),1001(authz)
root@titus:~/docker-authz# id titus
uid=1000(titus) gid=1000(titus) groups=1000(titus),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),126(sambashare),1001(authz)
root@titus:~/docker-authz# id pascal
uid=1003(pascal) gid=1004(pascal) groups=1004(pascal),27(sudo)
root@titus:~/docker-authz# id trugendo
uid=1001(trugendo) gid=1002(trugendo) groups=1002(trugendo),999(docker)
```

Figure 22: Host users’ groups

Figure 22 above shows the groups that each individual host system user belongs to. Only user ‘titus’ and ‘elly’ belong to ‘authz’ group. This means that they are the only ones with all privileges on the host system and that can access and modify the policy file.

```
trugendo@titus:~$ sudo bash
[sudo] password for trugendo:
trugendo is not in the sudoers file.  This incident will be reported.
trugendo@titus:~$ cat /usr/local/bin/docker-authz.py
cat: /usr/local/bin/docker-authz.py: Permission denied
trugendo@titus:~$ exit
logout
root@titus:/usr/local/bin# su - pascal
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

pascal@titus:~$ sudo bash
[sudo] password for pascal:
pascal is not in the sudoers file.  This incident will be reported.
pascal@titus:~$ cat /usr/local/bin/docker-authz.py
cat: /usr/local/bin/docker-authz.py: Permission denied
pascal@titus:~$ exit
logout
root@titus:/usr/local/bin# su - elly
elly@titus:~$ sudo bash
[sudo] password for elly:
root@titus:~# cat /usr/local/bin/docker-authz.py | head -n 3

from flask import Flask, jsonify, request
import base64, json
root@titus:~# id elly
uid=1002(elly) gid=1003(elly) groups=1003(elly),1001(authz)
root@titus:~#
root@titus:~#
```

Figure 23: Policy file read test

Figure 23 above shows different system users trying to read the policy file. User 'pascal' and 'trugendo' are denied read rights to the policy file because they do not belong to 'authz' group. User 'elly' can elevate to root mode and read the policy though he is not a member of the sudo group.

```
@plug.route("/AuthZPlugin.AuthZReq", methods=["POST"])
def req():
    plugin_request=json.loads(request.data)
    print(plugin_request)
    response={"Allow":True}
    if search(r'/(exec)$', plugin_request["RequestUri"]) != None:
        docker_request=json.loads(base64.b64decode(plugin_request["RequestBody"]))
        if match(r'^titus$|^elly$', docker_request["User"])!=None:
            response={"Allow":True}
        else:
            response={"Allow":False, "Msg":"You are not authorized to Run Execute command"}
    if not enabled:
        response={"Allow":True}
    return jsonify(**response)
```

Figure 24: Policy editing

Figure 24 shows policy being edited to allow user too ‘elly’ to access containers. According to policy any allowed user can access any container provided they exist in that container.

```
root@titus:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                NAMES
095bf6b1f4ff        e3faa30ac980        "docker-entrypoint.s…"   3 days ago          Up 16 minutes       5432/tcp             postgres_authorization_test
81a601d9d09a        c3502de97de8        "nginx -g 'daemon of…"   3 days ago          Up 15 minutes       0.0.0.0:80->80/tcp   nginx_authorization_test
root@titus:~# docker exec -it -u titus nginx_authorization_test bash
titus@81a601d9d09a:/$ ls /home
rugendo  titus
titus@81a601d9d09a:/$ exit
exit
root@titus:~# docker exec -it -u elly nginx_authorization_test bash
unable to find user elly: no matching entries in passwd file
^C
root@titus:~# docker exec -it -u titus postgres_authorization_test bash
titus@095bf6b1f4ff:/$ ls /home
elly  rugendo  titus
titus@095bf6b1f4ff:/$ exit
exit
root@titus:~# docker exec -it -u elly postgres_authorization_test bash
elly@095bf6b1f4ff:/$ id elly
uid=1002(elly) gid=1002(elly) groups=1002(elly),27(sudo)
elly@095bf6b1f4ff:/$ sudo bash
[sudo] password for elly:
root@095bf6b1f4ff:/#
```

Figure 25: New Policy Test

From figure 25 above, user ‘elly’ is not able to access Nginx container since he is not created in that container. Whereas he can access container postgres since he an existing user. This shows that the policy works for container users only. You must be a member of a certain container and allowed on the policy to be granted access.

Root User test

By default, container engine allows host users to access all containers in privileged mode. This allowed anyone from the host system to access the container and do any modifications as they wished. The developed access control plugin denies the default container root user access.

```
root@titus:~# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED      STATUS         PORTS                NAMES
095bf6b1f4ff   e3faa30ac980   "docker-entrypoint.s…"   3 days ago   Up 8 minutes   5432/tcp             postgres_authorization_test
81a601d9d09a   c3502de97de8   "nginx -g 'daemon of…"   3 days ago   Up 8 minutes   0.0.0.0:80->80/tcp   nginx_authorization_test
root@titus:~# docker exec -it -u root postgres_authorization_test bash
Error response from daemon: authorization denied by plugin docker-authz: You are not authorized to Run Execute command
root@titus:~# docker exec -it -u root  nginx_authorization_test bash
Error response from daemon: authorization denied by plugin docker-authz: You are not authorized to Run Execute command
root@titus:~#
```

Figure 26: Default container root user access request

From figure 26 above, the container root user is denied access to the container. This is because the policy allows only user 'titus' and 'elly' alone. Thus, all requests from root user will be denied.

Non-existing container user test

For this test, we are going to try and login to a docker container using a username that is not defined within the container. We will use one system user who is not in Nginx container. From this test we have determined that this plugin authorizes only container users and not host system users.

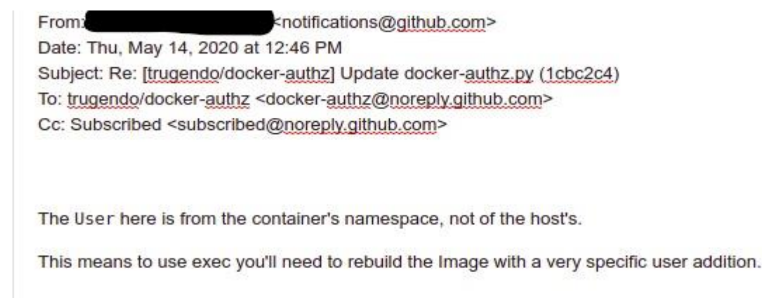
```
root@titus:~# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS         PORTS                NAMES
81a601d9d09a   c3502de97de8   "nginx -g 'daemon of…"   20 hours ago   Up 4 seconds   0.0.0.0:80->80/tcp   nginx_authorization_test
root@titus:~# ls /home
elly  pascal  titus  trugendo
root@titus:~# docker exec -it -u titus 81a601d9d09a bash
titus@81a601d9d09a:/$ ls /home
elly  rugendo  titus
titus@81a601d9d09a:/$ exit
exit
root@titus:~# docker exec -it -u pascal 81a601d9d09a bash
Error response from daemon: authorization denied by plugin docker-authz: You are not authorized to Run Execute command
root@titus:~# docker exec -it -u trugendo 81a601d9d09a bash
Error response from daemon: authorization denied by plugin docker-authz: You are not authorized to Run Execute command
root@titus:~#
```

Figure 27: Non-Container user test

From figure 27 above, users' 'pascal' and 'trugendo' are denied access to the Nginx container since they are not member of that container. Even though they are host system users, the policy makes allow or deny decisions based on container users. Thus, theirs requests will be treated like those of unknown users, leading to their requests being denied

4.1.6.3 Usability testing

The plugin code was shared with two Container virtualization administrators to test the usability of the docker authorization plugin and give a performance review.



From: [redacted] <notifications@github.com>
Date: Thu, May 14, 2020 at 12:46 PM
Subject: Re: [trugendo/docker-authz] Update docker-authz.py (1cbc2c4)
To: trugendo/docker-authz <docker-authz@noreply.github.com>
Cc: Subscribed <subscribed@noreply.github.com>

The User here is from the container's namespace, not of the host's.

This means to use exec you'll need to rebuild the Image with a very specific user addition.

Figure 28: Usability test Response

Figure 28 above shows the response from one container virtualization administrator who tested the plugin. He recommended that the plugin was best suitable for organization custom developed applications since you must create users in the images. For images downloaded from Docker hub, the administrator suggested that the container administrator had to create users then recreate another image for other deployments.

The second administrator input was that the plugin will help resolve a lot of issues regarding container access. Users who do configurations will be the only ones allowed privileged access to the containers. This will also simplify auditing since they will be checking container logs to determine which container user performed a certain action.

4.2 Discussion

The default access to container systems is based on all or nothing. Where you can either have full access or no access at all. In this study were able to find that you can control the access to container virtual environments by use of a plugin. The access control policy is defined in terms of users created in the container virtual environment rather than host users. This research led to development of an access control plugin model that uses container users to make decisions on who should be allowed access into a specific container. The users are created in the container by administrators and then used when making decisions based on the policy of the plugin. The plugin uses user information in the exec URL to check against the policy and container users before allowing access to

the container. Since the developed plugin can limit who has access to the containers, then a container administrator can give different file permissions and privileges to different container users. Where, sudo or superuser rights can be assigned to some specific users in the container and give other users access rights without super privileges. The user's actions on the containers will be depend on the rights and permissions they have once the plugin allows their access. The developed plugin does not make any decisions based on host system users; in fact, it denies all their access requests.

Existing container access control plugins work by allowing or denying users from running specific docker commands. Like for the case of authz plugin for docker containers by Everett Toews, you either run all commands or denied running any command at all. The authz plugin denies all requests until docker-machine root or any other user pulls and installs hello-world plugin image. After running this image all users in the docker-machine and host machine will be able to run all docker commands and even access containers without any restriction. This is like the default container access control policy of all or nothing access and running commands. Our developed access control plugin model denies only access requests but allows all other docker commands to be run by all users. To access a user must exist in the container and must be allowed in the plugin. This research resulting plugin does not allow any host user to access the container regardless of whichever situation.

In OPA container access control plugin, users used for access control are neither host system nor container users. You define users on the policy with read-only rights set to true or false and a json config file containing a specific user and http headers for sending the request to the Docker engine. If the user in the json file has read-only rights on the policy file, then you can only run read Docker commands. If the user read-only is set to false, then you can run all commands. The worry with this is that if the settings are using user with read-only set to false. You can run everything even gain access into the container as root user without being limited. This research resulting container access control plugin policy uses container users created by the container administrator to make container access decisions only. All other docker commands are not restricted to any user. Our plugin deals only with authorizing container access requests and does not make decisions for other requests.

The challenges of both OPA and authz container authorization plugins is that they do not make access decisions based on existing users. Either host system users or container users. Our plugin model addresses this by using container users created by container administrator to make decisions on who is allowed access. OPA and authz also do not restrict access into the containers rather they give authorizations in terms of commands one can run. If a user like in OPA can run all commands, then they can also access all containers in root mode. This shows that access decisions are not made depending on container users, rather if a certain user is set in the policy and json configuration file then all users on the hosts machine can access all containers in privileged mode. Our plugin has addressed this by making sure that access to containers is based on container users only. A user who is not created in a certain container cannot access it even if they have been allowed on the policy. Also, users created by the container administrators for our case and are allowed access by the plugin, they gain access as unprivileged users. Thus, a user can only perform actions like write, execute or modification of files if they are allowed privileged rights. This means that we can allow multiple users to the container and still limit what they can do within the container environment.

The limitation of our plugin compared to OPA and authz is that we are only concerned with access requests to containers, but we are not authorizing other container requests that are run by host system users. The similarity issue with all these models is there is a weakness in securing the policy document. Like for this plugin we had to disable sudo and admin group privileges to protect the file against access by unauthorized system privileged users. If there are other applications running in the host system, the policy file access will need to depend on trust security mechanism.

From tests on the developed plugin, it shows that this research has led to the development of a better access control plugin that makes access decisions based on who is allowed into the containers. It has brought another level of access control to containers where decisions are made based on container users and not host users. No user is directly allowed into the container as root user. The main challenge during access is that the allowed users are only authenticated against usernames only. The person is not asked for password when the request is allowed. Though they gain access in unprivileged and only

elevated to superuser mode by authenticating using a password. This means we have also to protect the policy file on the host machine even more from any unauthorized access.

CHAPTER 5

5.0 Conclusion and Recommendations

5.1 Conclusion

In this research, we sought to review the concept of access control and, try to address the challenge of access to container-based environments by all privileged users in a host system. We were able to evaluate and determine how current container virtualization authorization frameworks implement access control. And, how they interact with container engines. Additionally, we were able to design and develop an efficient plugin for controlling access to containers. This study has successfully established that, a container access control model that makes access decisions based on container specific users can be implemented.

Our first objective was to review the concept of access control in container virtual computing environments. This study found out that the default container virtual environment access control is based on all or nothing mechanism. Where all privileged host system users have full access and all privileges on all containers while host system non privileged users have no access at all. To address this, container technology allows third party plugins to be integrated with them to achieve additional features. Container developers have created syntaxes to be used to ensure proper integration of the plugin with the container engine. One common syntax provided is that of an access control plugin, especially for Docker containers. Generally, container engine communicates with plugins using generic APIs and HTTP methods. For the container engine to discover the plugins they must be added to a plugin directory that is specified by a given container technology. And, a policy which is a decision point that must be defined. Where all access requests must pass through, then a decision is made whether access is granted or denied.

Our second objective was to evaluate the current container authorization frameworks in terms of how they implement access control. From this study we found out that several container authorization plugins have been developed to help implement access control in Docker containers. The common one’s included OPA and authz. Currently none of the two makes decisions based on users within the container or host

system users. The authz plugin denies all request until hello-world docker image is run by a single privileged host system user. Once, the hello-world starts running all users in the host machine can run all commands which had been restricted before, and even access all containers as root. OPA defines users within its policy and gives them access rights based on container commands. The access rights are defined in terms of read-only, where a user with read only access cannot run write docker commands. While a user with all rights can run all docker commands and even access the containers without any restrictions. OPA allows one to use a single user account at a time.

Our third objective was to design and develop an access control plugin model that will make access decisions to containers based on virtual environment users. This research led to the design and development of a container access control plugin that uses a policy and container specific users to determine which user is allowed access to the container environment. The development was based on the default syntax provided by Docker for developing a plugin.

Our final objective was to test and evaluate the performance and effectiveness of the developed access control plugin. By tests done using our plugin, we have successfully showed that access into containers can be controlled by using container specific users. We were able to deny all host system users access to containers despite their privileges. And, we were able to deny the default container root access to all containers. This enables the administrator to give different container users different privileges and rights within the containers.

5.2 Limitations for this study

This plugin can work best on a host system that is running all applications on containers. If we have some applications running on the host system and having multiple groups with privileged rights, the policy security might be threatened. This is because the policy file is stored within the host system.

The developed container access control plugin can only be used to restrict access to containers based on users in the specific container virtual environment. A container administrator must first create specific container users and give them different privileges since, all host system users are denied access to the containers.

Our plugin can only be implemented using Docker containers. Currently it cannot be used with other container technologies since it is based on the Docker syntax of communication. Where communication between Docker engine and the plugin uses HTTP methods and docker .spec file.

5.3 Recommendations for further work

Container developers like docker and LXC should come up with a way of prompting authorized users for passwords as they access the containers. Currently container engines do not provide a way of including password in the URL or requesting for password after being allowed access to the containers. The plugin uses only username to authenticate users who are requesting access. No password is required. Though the users are given access in unprivileged mode it would be nice to authenticate users using password before access. From the research we have seen that you must add privileges to specific users in the container like sudo rights where they can authenticate using a password to execute commands. Whereas they are not asked for a password when they are trying to access the containers.

Currently, our plugin can work on a host system that is running all applications on containers. If we have some applications running on the host system and having multiple groups with privileged rights, the policy file security might be threatened. This is because our policy file is located within the host Operating System. Future studies should try to find a way of incorporating the policy file within a specific container.

Future works should also look at how a container access control plugin can be interfaced with host system users. This will help address the issue of which system user can run a specific container command.

Finally, future works should try to implement an access control plugin that can be used with other container technologies.

References

Alenius, F. (2010). *Authentication and Authorization: Achieving Single Sign-on in an Erlang Environment.* Uppsala: Uppsala University.

Bacis, E., Mutti, S., Capelli, S., & Paraboschi, S. (2015). DockerPolicyModules: Mandatory Access Control for Docker Containers. *IEEE Conference on Communications and Network Security.* Bergamo: unibg.it. doi:10.1109/CNS.2015.7346917

Block, A., & Spazzoli, R. (2019, February 26). *Fine-Grained Policy Enforcement in OpenShift with Open Policy Agent*. Retrieved December 30, 2019, from Red Hat OpenShift: https://blog.openshift.com/fine-grained-policy-enforcement-in-openshift-with-open-policy-agent/

Bui, T. (2015, February 11). *Analysis of Docker Security.* Aalto University.

Chelladhurai, J., Chelliah, P. R., & Kumar, S. A. (2016). Securing Docker Containers from Denial of Service. *IEEE International Conference on Services Computing* (pp. 1-4). San Francisco, CA, USA: IEEE.

docker Inc. (2019). *docker docs*. Retrieved February 04, 2020, from docs.docker.com: https://docs.docker.com/engine/extend/plugins_authorization/

docker Inc. (2019). *Docker v19.03*. Retrieved March 20, 2020, from docker docs: https://docs.docker.com/engine/extend/plugin_api/

Good, M. (2018, July 23). *Intro to Attribute Based Access Control (ABAC)*. Retrieved December 30, 2019, from axiomatics: https://www.axiomatics.com/blog/intro-to-attribute-based-access-control-abac/

Hauser, F., Schmidt, M., & Menth, M. (2019). xRAC: Execution and Access Control for Restricted Application Containers on Managed Hosts. *ArXiv, abs/1907.03544*, 1-9.

Hu, V. C., Ferraiolo, D., Kuhn, R., Schnitzer, A., Sandlin, K., Miller, R., & Scarfone, K. (2014). *Guide to Attribute Based Access Control (ABAC) Definition and Considerations.* Gaithersburg: National Institute of Standards and Technology.

Jain, H. (2016, June 2). *LXC and LXD: Explaining Linux Containers*. Retrieved April 27, 2020, from Sumo Logic: https://www.sumologic.com/blog/lxc-lxd-linux-containers/

Kenlon, S. (2020, January 30). *Exploring simple Linux containers with lxc*. Retrieved April 27, 2020, from Red Hat: https://www.redhat.com/sysadmin/exploring-containers-lxc

Kuusik, K. (2015, June 19). *Docker Security – Admin Controls*. Retrieved January 12, 2020, from Container Solutions: https://blog.container-solutions.com/docker-security-admin-controls-2

Lang, D., Jiang, H., Ding, W., & Bai, Y. (2019). Research on Docker Role Access Control Mechanism Based on DRBAC. *Journal of Physics: Conference Series. 1168*, pp. 1-9. Beijin: IOP Publishing Ltd.

Levin, L. (2016, February 18). *Docker AuthZ Plugins: Twistlock's Contribution to Docker Security*. Retrieved December 29, 2019, from Twistlock: https://www.twistlock.com/2016/02/18/docker-authz-plugins/

Nosek, A. (2019, October 19). *Open Policy Agent, Part I - The Introduction* . Retrieved December 30, 2019, from DZone: https://dzone.com/articles/open-policy-agent-part-i-the-introduction

Pahl, C., Brogi, A., Soldani, J., & Jamshidi, P. (2017, May). Cloud Container Technologies: a State-of-the-Art Review. *IEEE Transactions on Cloud Computing*, 1. doi:10.1109/TCC.2017.2702586

Rodriguez, M., & Buyya, R. (2018). Container-based Cluster Orchestration Systems: A Taxonomy and Future Directions. *Distributed, Parallel, and Cluster Computing*, 1-29.

Shoeb, Z. H., & Sobhan, A. (2010). Authentication and Authorization: Security Issues for Institutional Digital Repositories. *Library Philosophy and Practice*, 1-8.

Sultan, S., Ahmad, I., & Dimitriou, T. (2019, April 17). Container Security: Issues, Challenges, and the Road Ahead. *IEEE Access, vii*, 52976 - 52996.

Toews, E. (2016, July 30). *Develop a Docker Authorization Plugin in Python*. Retrieved February 20, 2020, from github: https://etoews.github.io/blog/2016/07/30/develop-a-docker-authz-plugin-in-python/

Victor, G. d., Marite, K., & Gundars, A. (2018, May). Containers for Virtualization: An Overview. *Applied Computer Systems , XXIII*, 21-27.

Appendices

Project Schedule

Sample Code:

```
@plug.route("/info/<query>", methods=["GET"])
def state(query):
if query=="state":
qu=(1 if enabled else 0)
return str(qu)
else:
return "-1"

@plug.route("/Plugin.Activate", methods=["POST"])
def start():
return jsonify({"Implements": ["authz"]})

@plug.route("/AuthZPlugin.AuthZReq", methods=["POST"])
```

```
def req():
plugin_request=json.loads(request.data)
print(plugin_request)
response={"Allow":True}
if search(r'/(exec)$', plugin_request["RequestUri"]) != None:
docker_request=json.loads(base64.b64decode(plugin_request["RequestBody"]))
if match(r'^titus$', docker_request["User"])!=None:
response={"Allow":True}
else:
response={"Allow":False, "Msg":"You are not authorized to Run Execute command"}
if not enabled:
response={"Allow":True}
return jsonify(**response)

@plug.route("/AuthZPlugin.AuthZRes", methods=["POST"])
def res():
response={"Allow":True}
return jsonify(**response)
```

Guide on how to use the container access control Plugin (docker-authz):

Installing the plugin.

The plugin was deployed on a host machine running Ubuntu 18.04 LTS and had Docker engine version 19.03.8 installed. To deploy the plugin run the install.sh script to update the system then to install the requirements for this plugin to run. The requirements are:

i. python3

ii. pytthon3-Flask

iii. jq

Enter the directory containing the plugin code and change the scripts and the python code to executable mode. Using command chmod +x.

Changing python and bash scripts to executable

```
root@titus:~/docker-authz# ls
authz.json  docker-authz.py  docker-authz.service  docker-authz.sh  docker-authz.spec  install.sh  README.md  requirements.txt
root@titus:~/docker-authz# chmod +x docker-authz.py docker-authz.sh install.sh
root@titus:~/docker-authz# ls
authz.json  docker-authz.py  docker-authz.service  docker-authz.sh  docker-authz.spec  install.sh  README.md  requirements.txt
```

Exit from privileged mode to run the installation script.

```
titus@titus:~/docker-authz$ ./install.sh
==> Updating system
[sudo] password for titus:
==> Installing updates
Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
The following packages were automatically installed and are no longer required:
```

Installing plugin

Finalizing installation

```
done
==> System update successful
==> Installing dependencies
[sudo] password for titus:
```

```
==> Dependencies satisfied
==> Creating necessary directories
==> Directory creation succeessful
==> Installation successful
==> Post installation jobs
 - stopping docker
 - starting docker & docker-authz
Created symlink /etc/systemd/system/multi-user.target.wants/docker-authz.service → /lib/systemd/system/docker-authz.service.
==> Post install processes are now finished.
titus@titus:~/docker-authz$
```

The policy and startup script for enabling and disabling the plugin are installed at location. */usrlocal/bin/.*

Policy location

```
titus@titus:~/docker-authz$ ls /usr/local/bin/
docker-authz  docker-authz.py  flask
```

Operating docker authorization Plugin

The plugin starting and stopping can be done from systemd. Also, one can view the status of the plugin, if it is running or if it is off. The plugin needs to be restarted whenever any changes are made to the policy to ensure the new changes take effect. You need to have super user privileges to perform start, stop or restart operations.

One can use below commands to start, stop, restart and view status of the plugin

Systemctl start docker-authz or service docker-authz start

Systemctl stop docker-authz or service docker-authz stop

Systemctl status docker-authz or service docker-authz status

Systemctl restart docker-authz or service docker-authz restart

Plugin Operations

```
May 17 00:05:44 titus systemd[1]: Started docker-authz.
May 17 00:05:54 titus python3[1082]:  * Serving Flask app "docker-authz" (lazy loading)
May 17 00:05:54 titus python3[1082]:  * Environment: production
May 17 00:05:54 titus python3[1082]:    WARNING: This is a development server. Do not use it in a production deployment.
May 17 00:05:54 titus python3[1082]:    Use a production WSGI server instead.
May 17 00:05:54 titus python3[1082]:  * Debug mode: off
May 17 00:05:54 titus python3[1082]:  * Running on http://127.0.0.1:6000/ (Press CTRL+C to quit)
May 17 00:06:10 titus python3[1082]: 127.0.0.1 - - [17/May/2020 00:06:10] "POST /Plugin.Activate HTTP/1.1" 200 -
root@titus:~# systemctl stop docker-authz
root@titus:~# systemctl status docker-authz
● docker-authz.service - docker-authz
   Loaded: loaded (/lib/systemd/system/docker-authz.service; enabled; vendor preset: enabled)
   Active: inactive (dead) since Sun 2020-05-17 00:07:18 EAT; 2s ago
  Process: 1082 ExecStart=/usr/bin/python3 /usr/local/bin/docker-authz.py (code=killed, signal=TERM)
 Main PID: 1082 (code=killed, signal=TERM)

May 17 00:05:44 titus systemd[1]: Started docker-authz.
May 17 00:05:54 titus python3[1082]:  * Serving Flask app "docker-authz" (lazy loading)
May 17 00:05:54 titus python3[1082]:  * Environment: production
May 17 00:05:54 titus python3[1082]:    WARNING: This is a development server. Do not use it in a production deployment.
May 17 00:05:54 titus python3[1082]:    Use a production WSGI server instead.
May 17 00:05:54 titus python3[1082]:  * Debug mode: off
May 17 00:05:54 titus python3[1082]:  * Running on http://127.0.0.1:6000/ (Press CTRL+C to quit)
May 17 00:06:10 titus python3[1082]: 127.0.0.1 - - [17/May/2020 00:06:10] "POST /Plugin.Activate HTTP/1.1" 200 -
May 17 00:07:18 titus systemd[1]: Stopping docker-authz...
May 17 00:07:18 titus systemd[1]: Stopped docker-authz.
root@titus:~# systemctl start docker-authz
root@titus:~# systemctl status docker-authz
● docker-authz.service - docker-authz
   Loaded: loaded (/lib/systemd/system/docker-authz.service; enabled; vendor preset: enabled)
   Active: active (running) since Sun 2020-05-17 00:07:28 EAT; 2s ago
 Main PID: 2640 (python3)
    Tasks: 1 (limit: 2327)
   CGroup: /system.slice/docker-authz.service
           └─2640 /usr/bin/python3 /usr/local/bin/docker-authz.py

May 17 00:07:28 titus systemd[1]: Started docker-authz.
May 17 00:07:28 titus python3[2640]:  * Serving Flask app "docker-authz" (lazy loading)
May 17 00:07:28 titus python3[2640]:  * Environment: production
May 17 00:07:28 titus python3[2640]:    WARNING: This is a development server. Do not use it in a production deployment.
May 17 00:07:28 titus python3[2640]:    Use a production WSGI server instead.
May 17 00:07:28 titus python3[2640]:  * Debug mode: off
May 17 00:07:28 titus python3[2640]:  * Running on http://127.0.0.1:6000/ (Press CTRL+C to quit)
root@titus:~#
```

Securing Policy

The plugin consists of:

i. The policy – where decisions on who has access to the containers is made.

ii. A bash script to enable or disable the authorization plug in. The commands can only be executed by super users in authz group only.

The default sudo and admin group have been disabled in file /etc/sudoers/ and group authz has been added to the file with all privileges. Note that host users in docker group can run docker commands in unprivileged mode. The permissions for bash script and policy in location /usr/local/bin were changed so as to allow only users in authz group to access and modify the files. This has been done to prevent unauthorized users from editing the policy and the enabling and disabling bash script. The group authz is created and protected by an encrypted password.

Creating authz group

```
root@titus:~/docker-authz# groupadd -p Auth@Z authz
```

Assigning authz group users

```
#
# This file MUST be edited with the 'visudo' command as root.
#
# Please consider adding local content in /etc/sudoers.d/ instead of
# directly modifying this file.
#
# See the man page for details on how to write a sudoers file.
#
Defaults        env_reset
Defaults        mail_badpass
Defaults        secure_path="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin"

# Host alias specification

# User alias specification

# Cmnd alias specification

# User privilege specification
root    ALL=(ALL:ALL) ALL

# Members of the admin group may gain root privileges
#%admin ALL=(ALL) ALL
%authz ALL=(ALL) ALL
# Allow members of group sudo to execute any command
#%sudo  ALL=(ALL:ALL) ALL

# See sudoers(5) for more information on "#include" directives:

#includedir /etc/sudoers.d
root@titus:/usr/local/bin#
root@titus:/usr/local/bin#
```

The files ownership and group ownership were changed to user 'titus' as the owner and group authz as the owner group. The read, write and execute commands were enabled for owner and owner group and denied for other users using below commands:

chown titus docker-authz.py docker-authz – assigns the policy file to user titus who is the administrator from default user root.

chgrp authz docker-authz.py docker-authz – assigns the policy file to the authz group.

chmod 770 docker-authz.py docker-authz – assigns read, write and execute permissions to the policy fire for all members of authz group

Changing policy file ownership and permissions

```
root@titus:~/docker-authz# cd /usr/local/bin/
root@titus:/usr/local/bin# ls
docker-authz  docker-authz.py  flask
root@titus:/usr/local/bin# ls -lrth
total 16K
-rwxr-xr-x 1 root root  212 Cam 11 19:16 flask
-rwx------ 1 root root 2.1K Cam 16 13:31 docker-authz.py
-rwx------ 1 root root 7.1K Cam 16 13:31 docker-authz
root@titus:/usr/local/bin# chown titus docker-authz.py
root@titus:/usr/local/bin# chown titus docker-authz
root@titus:/usr/local/bin# chgrp authz docker-authz.py
root@titus:/usr/local/bin# chgrp authz docker-authz
root@titus:/usr/local/bin# ls -lrth
total 16K
-rwxr-xr-x 1 root  root   212 Cam 11 19:16 flask
-rwx------ 1 titus authz 2.1K Cam 16 13:31 docker-authz.py
-rwx------ 1 titus authz 7.1K Cam 16 13:31 docker-authz
root@titus:/usr/local/bin# chmod 770 docker-authz.py docker-authz
root@titus:/usr/local/bin# ls -lrth
total 16K
-rwxr-xr-x 1 root  root   212 Cam 11 19:16 flask
-rwxrwx--- 1 titus authz 2.1K Cam 16 13:31 docker-authz.py
-rwxrwx--- 1 titus authz 7.1K Cam 16 13:31 docker-authz
```

Four users on the host machine were added to different groups. Two users to authz group (titus and elly) and one in docker group (trugendo) and one in sudo group (pascal). The user on docker group is expected to run all docker commands but not able to access or edit the policy file. The user in sudo group only will not be able to run any docker commands and will not be able to access the policy file. The two users in authz group will be able to access and edit the policy file.

```
root@titus:~/docker-authz# ls /home/
elly  pascal  titus  trugendo
root@titus:~/docker-authz# id elly
uid=1002(elly) gid=1003(elly) groups=1003(elly),1001(authz)
root@titus:~/docker-authz# id titus
uid=1000(titus) gid=1000(titus) groups=1000(titus),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),126(sambashare),1001(authz)
root@titus:~/docker-authz# id pascal
uid=1003(pascal) gid=1004(pascal) groups=1004(pascal),27(sudo)
root@titus:~/docker-authz# id trugendo
uid=1001(trugendo) gid=1002(trugendo) groups=1002(trugendo),999(docker)
```

Adding host system users to groups

User pascal who is in sudo group cannot get to privilleged mode since sudo group has been disabled on sudoers file. User elly can be elevated to privilleged mode though not in sudo group since he his in authz group which has been granted all privileges in sudoers file. User trugendo and pascal are not able to access the policy file since they are not in authz group.

```
trugendo@titus:~$ sudo bash
[sudo] password for trugendo:
trugendo is not in the sudoers file.  This incident will be reported.
trugendo@titus:~$ cat /usr/local/bin/docker-authz.py
cat: /usr/local/bin/docker-authz.py: Permission denied
trugendo@titus:~$ exit
logout
root@titus:/usr/local/bin# su - pascal
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

pascal@titus:~$ sudo bash
[sudo] password for pascal:
pascal is not in the sudoers file.  This incident will be reported.
pascal@titus:~$ cat /usr/local/bin/docker-authz.py
cat: /usr/local/bin/docker-authz.py: Permission denied
pascal@titus:~$ exit
logout
root@titus:/usr/local/bin# su - elly
elly@titus:~$ sudo bash
[sudo] password for elly:
root@titus:~# cat /usr/local/bin/docker-authz.py | head -n 3

from flask import Flask, jsonify, request
import base64, json
root@titus:~# id elly
uid=1002(elly) gid=1003(elly) groups=1003(elly),1001(authz)
root@titus:~#
root@titus:~#
```

testing docker-authz policy access permissions

Setting Up Docker working Environment

For this test we created two docker containers running Nginx and PostgreSQL images.The containers were edited by adding three users in each image (titus, rugendo and elly). Then the containers were committed to new images called; nginx_authorization_test and postgres__authorization_test. User titus has sudo rights on both images. User elly has super user privileges only on the postgres database image.

Pulling test images

Nginx and Postgres images were pulled from docker hub. Nginx is set to run on port 80 and postgres is running on port 5432. The ‘docker pull’ command was used. After pulling images from Docker hub, they are stored in the local pool of the docker engine. From here you can create containers from these images and edit with desired settings and data, then recreate another image with all information in it.

Testing docker-authz policy access permissions

```
root@titus:~# docker run --name nginx_auth_test -p 80:80 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
afb6ec6fdc1c: Pull complete
b90c53a0b692: Pull complete
11fa52a0fdc0: Pull complete
Digest: sha256:30dfa439718a17baafefadf16c5e7c9d0a1cde97b4fd84f63b69e13513be7097
Status: Downloaded newer image for nginx:latest
^C^Croot@titus:~#
root@titus:~#
root@titus:~# docker pull postgres
Using default tag: latest
latest: Pulling from library/postgres
afb6ec6fdc1c: Already exists
51be5f829bfb: Pull complete
e707c08f571a: Pull complete
98ddd8bce9b5: Pull complete
5f16647362a3: Pull complete
5d56cdf9ab3b: Pull complete
2207a50ca41d: Pull complete
a51d14a628f3: Pull complete
24dcb11335d0: Pull complete
54cc759cb0bb: Pull complete
debc11d66570: Pull complete
3ffd0589b5fc: Pull complete
490b7ee49751: Pull complete
3511c6be34a0: Pull complete
Digest: sha256:dcbd89d8a92f9f8afefc474df7bdca7d37e4cb1e6e4cc88dc42579fc26d1db53
Status: Downloaded newer image for postgres:latest
docker.io/library/postgres:latest
root@titus:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
postgres            latest              adf2b126dda8        6 hours ago         313MB
nginx               latest              9beeba249f3e        13 hours ago        127MB
root@titus:~# docker ps -a
```

Image editing and recreation

```
root@titus:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
a27aced14c81        postgres            "docker-entrypoint.s…"   3 minutes ago       Up 3 minutes                5432/tcp            sleepy_hermann
6cea82374b3e        nginx               "nginx -g 'daemon of…"   20 minutes ago      Exited (0) 19 minutes ago                       nginx_auth_test
root@titus:~# docker start 6cea82374b3e
6cea82374b3e
root@titus:~# docker start a27aced14c81
a27aced14c81
root@titus:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                NAMES
a27aced14c81        postgres            "docker-entrypoint.s…"   4 minutes ago       Up 3 minutes        5432/tcp             sleepy_hermann
6cea82374b3e        nginx               "nginx -g 'daemon of…"   21 minutes ago      Up 13 seconds       0.0.0.0:80->80/tcp   nginx_auth_test
```

The two images were run to containers then started. The running containers accessed using root mode, by running command 'docker exec'. The three users were created in each container. Privilege rights being given to specific users (titus in both containers and elly in postgres container only). Once the users have been created, the containers will be committed to new images that contain the new users. These images are stored in the docker engine but can also be pushed to docker hub or transferred to other hosts for use with the new dat

.

```
root@titus:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED              STATUS              PORTS                NAMES
ca6b3cb57f1d        adf2b126dda8        "docker-entrypoint.s…"   About a minute ago   Up 7 seconds        5432/tcp             postgres_auth_test
6cea82374b3e        nginx               "nginx -g 'daemon of…"   30 minutes ago       Up 4 minutes        0.0.0.0:80->80/tcp   nginx_auth_test
root@titus:~# docker exec -it ca6b3cb57f1d bash
root@ca6b3cb57f1d:/# adduser titus
Adding user `titus' ...
Adding new group `titus' (1000) ...
Adding new user `titus' (1000) with group `titus' ...
Creating home directory `/home/titus' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for titus
Enter the new value, or press ENTER for the default
        Full Name []: titus
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n] y
root@ca6b3cb57f1d:/# adduser rugendo
Adding user `rugendo' ...
Adding new group `rugendo' (1001) ...
Adding new user `rugendo' (1001) with group `rugendo' ...
Creating home directory `/home/rugendo' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for rugendo
```

Creating Docker containers from pulled Nginx and PostgreSQL images and Starting them

```
root@6cea82374b3e:/# hostname
6cea82374b3e
root@6cea82374b3e:/# id titus
uid=1000(titus) gid=1000(titus) groups=1000(titus),27(sudo)
root@6cea82374b3e:/# id rugendo
uid=1001(rugendo) gid=1001(rugendo) groups=1001(rugendo)
root@6cea82374b3e:/# id elly
uid=1002(elly) gid=1002(elly) groups=1002(elly)
```

Checking container users privileges

From above we can see Nginx users. Only user 'titus' has privileged rights in group sudo.

Testing docker-authz Plugin access control functionality.

Policy is defined as below. Only user 'titus' from within the container can access the containers using 'exec' command.

```
    if search(r'/(exec)$', plugin_request["RequestUri"]) != None:
        docker_request=json.loads(base64.b64decode(plugin_request["RequestBody"]))
        if match(r'^titus$', docker_request["User"])!=None:
            response={"Allow":True}
        else:
            response={"Allow":False, "Msg":"You are not authorized to Run Execute command"}
    if not enabled:
        response={"Allow":True}
    return jsonify(**response)

@plug.route("/AuthZPlugin.AuthZRes", methods=["POST"])
def res():
    response={"Allow":True}
    return jsonify(**response)
```

Defined docker-authz policy

The access control test was done on docker containers running on docker daemon 19.03.8. The two containers were started and access to containers was tested by running 'docker exec' command.

Starting Containers from Pre-saved edited images

```
root@titus:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED              STATUS                      PORTS               NAMES
095bf6b1f4ff        e3faa30ac980        "docker-entrypoint.s…"   About a minute ago   Exited (0) 9 seconds ago                        postgres__authorization_test
81a601d9d09a        c3502de97de8        "nginx -g 'daemon of…"   5 minutes ago        Exited (0) 2 minutes ago                        nginx_authorization_test
ca6b3cb57f1d        adf2b126dda8        "docker-entrypoint.s…"   3 hours ago          Exited (0) 3 hours ago                          postgres_auth_test
6cea82374b3e        nginx               "nginx -g 'daemon of…"   4 hours ago          Exited (0) 3 hours ago                          nginx_auth_test
root@titus:~# docker start 81a601d9d09a
81a601d9d09a
root@titus:~# docker start 095bf6b1f4ff
095bf6b1f4ff
root@titus:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                NAMES
095bf6b1f4ff        e3faa30ac980        "docker-entrypoint.s…"   2 minutes ago       Up 3 seconds        5432/tcp             postgres__authorization_test
81a601d9d09a        c3502de97de8        "nginx -g 'daemon of…"   5 minutes ago       Up 12 seconds       0.0.0.0:80->80/tcp   nginx_authorization_test
```