Access Control Model for Container based Virtual Environments

Titus Murithi Rugendo
School of Computing and Informatics
University of Nairobi, Kenya

Andrew Mwaura Kahonge
School of Computing and Informatics
University of Nairobi, Kenya

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 root user, with all privileges and 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 analyses existing container authorization plugins to determine how they make access decisions. Additionally, this study led to the design and development of an effective access control plugin that makes access decisions to containers based on container users.

Keywords
Virtualization, Container, Docker, Access Control, Authorization

1. INTRODUCTION
Virtualization technology is becoming widely adopted since the last decade [1]. It involves partitioning a computer system into multiple isolated virtual environments. Various virtualization solutions have emerged to the market. These solutions 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 paper focuses on access control in container virtualization technology. There are several container technologies that are currently available and in use, namely LXC, OpenVZ, Linux-Vserver and Docker, with Docker being the most predominant. LXC, Linux Container was the origin of container revolution. It was also used as the underlying technology when implementing LXD and Docker containers [2]. This paper is going to focus on Docker container since it is the most widely adopted container technology 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 [1].

Access control is a key part of securing a computer system. It prevents illegal and legal entities from illegally accessing authorized resources. With increased used of virtual environments, many security concerns are rising and need to be addressed. Hypervisor based virtual environments are considered more secure than container based virtual environments since they provide an extra isolation layer between the host and the applications. Access to hypervisor based virtual environments is limited to users within the guest Operating System. In container based virtual environments, any user with privileged access rights on the host machine, especially in Linux kernel can access all container environments as super users without being authenticated. This means that all users with administrator rights can access and modify contents and applications running within different containers even if they are not supposed to [3]. This is a security challenge that this study is trying to address. Figure 1 below shows how container virtual environments are structured.

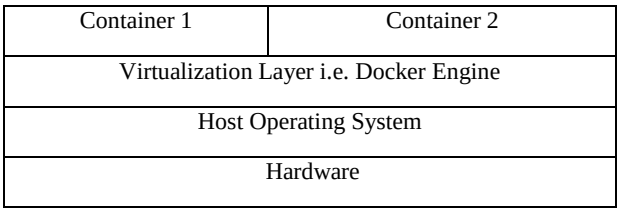

Figure 1: Container-based Virtualization Architecture

Containerization is a lightweight form of virtualization that consumes less space and time to start. 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 [4].

Access control [5] in container virtual environments can be achieved by extending container functionalities using plugins communicating with the container engine. For Docker containers they have 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 [6].

Ideally access control is supposed to prevent illegal access to unauthorized resources. However, currently all privileged users within a host Linux Operating system can gain root access, with full privileges to all containers running on that host without requiring any form of authorization. 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 [7].

The objectives for this study include: First, to review the concept of access control in container virtual computing environments. Secondly, to evaluate the existing container authorization frameworks in terms of how they implement access control. Thirdly, to design and develop an access control plugin model that will make access decisions to containers based on a specific virtual environment user. And, finally to test and evaluate the performance and effectiveness of the developed access control plugin in container based virtual environments.

2. RELATED WORK

There is already an existence of several container virtualization authentication plugins that have been developed. These plugins are used to perform authentication in Docker containers. This is because Docker has provided a way of extending its functionalities using external plugins. Docker engine allows the use of HTTP methods to communicate with the plugins. The plugins must also be stored in designated directories, to be discovered by the Docker engine. There are only three types of files that can be put in a plugin directory [8].

- . json – files containing full json specification for the plugin.
- . sock – UNIX domain sockets.
- .spec – files containing URLs, like tcp://localhost:port_number

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 [9].

When an access request is made by the user the Docker daemon passes the request to the installed access control plugin. The plugin is responsible for making the decision whether the user is allowed or not allowed to access or run a certain Docker command. For an access control plugin to communicate with the container engine, request syntax should include: User, Request URI, Request Method, Request Body and Request header. And, the response allowed is of the syntax: Allow which is a Boolean value of either true or false, Message and Error if any [8]. Figure 2 below shows how a Docker engine should interact with an authorization plugin to allow a user to perform actions they are authorized to.

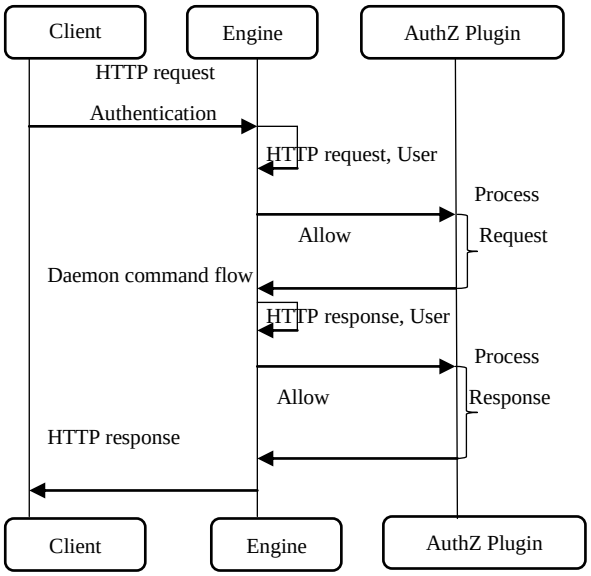

Figure 2: Authorization allow

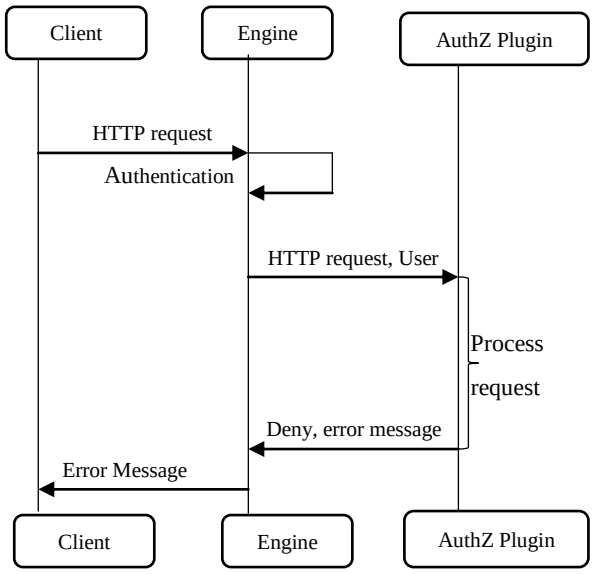

Figure 3: Authorization deny

Figure 3 above shows how the Docker engine should interact with an authorization plugin to deny a user request from performing an action they are not authorized to.

The two common Docker authentication plugins are Open Policy agent and Authz by Everett Toews. The two plugin technologies have been implemented to authorize what Docker commands a certain user or all users can run.

2.1 Open Policy agent (OPA)

OPA uses TLS to allow the Docker engine to authenticate users. OPA is uses three inputs to make authentication decisions [10].

- Data – Which is a set of facts about the outside world. This could be a list of users and their granted permissions.
- 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.

- 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.

The users being authenticated are already predefined within a file that will be stored in a certain location whose path will be added to the plugin for decision making. The permissions too are predefined and tied to a specific user in another file that is also linked to the plugin. Thus, OPA plugin makes authorization decisions based on its defined users and not host machine users or specific container users. The access decisions and permissions defined only affect the docker commands that a defined user can run. If a user is set with write permissions, then they can be able to access all the containers has the default privileged user.

2.2 Docker-authz-plugin

This plugin was developed by Everett Toews, it provides all or nothing authorization mechanism. Where all host system users can run all commands and even access containers as privileged users or not being able to access and run a single container command for all system users. Initially the plugin denies all authorization until any system user runs the hello world docker image, then access for all users is allowed [11].

By default, Docker containers employ an all or nothing access control mechanism. Since Docker is based on Linux kernel it uses Linux default autonomous access control technology. Where access to containers is achieved by adding roles and permissions to host system users [12]. That allows all privileged host system users to access the containers has root user, with all privileges. And, denies all non-privileged host users' access to containers. Since the default engine cannot make access control decisions [6], plugins like the ones discussed above are being developed to address this issue.

3. PROPOSED MODEL

The container administrator will be responsible for creating 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.

All host system users' access to containers will not be allowed by the plugin. The default root container access with all privileges within the container will also not be allowed by the plugin. Also, allowed containers users will only be able to access the containers in unprivileged mode. Only container users with privileged mode should be able to elevate themselves to have super user rights within the containers. Changes within a container can only be performed by privileged users only. Unprivileged users can only perform read operations.

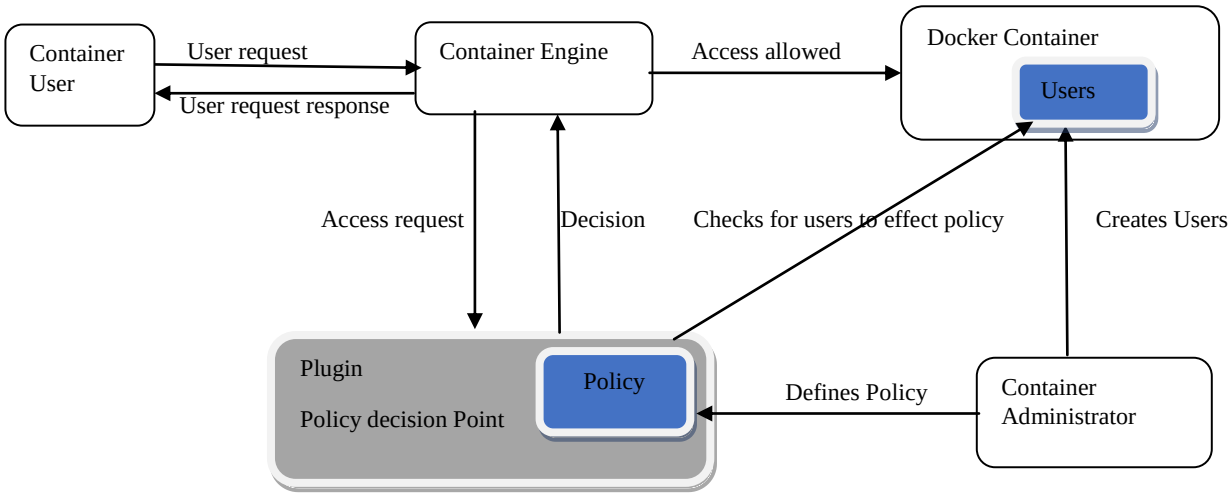

Figure 4: The Proposed Model

Figure 4 above illustrates how the plugin will operate with the Docker daemon or engine to control access to different containers. The access decision is based on users created within the containers and if they are allowed access to the container within the policy file. Note that all users allowed access to containers by policy access the containers in unprivileged mode since the default root access will be disabled by the plugin. Actions that users can perform within the containers will be determined by privileges given by the container administrator.

4. METHODOLOGY

The research was based on exploratory research design, it was divided into two parts: The first part involved reviewing literature, whereby existing materials in the container virtualization field were reviewed. The aim for this was to develop a strong background and determine whether there are other container access control plugins and how they have been implemented. Also, to determine how container engines communicate with external plugins to implement additional features that cannot be achieved by the container on its own.

The second part involved exploring current container access control plugins by performing numerous tests. The tests were meant to show how the plugins communicate with the container engines. And, how they implement access control or authorization for containers. This led to the identification of the current gaps in container access control.

Data for this study was collected using focus group discussions and observations. The focus group involved a discussion with five container virtualization experts within Nairobi, Kenya. The discussion was aimed at identifying the common themes among the existing container technologies in terms of access control. Observations involved testing the existing container authorization plugins and determining how they interact with the container engine to achieve authorization.

The main aim of this research is to develop an efficient container access control plugin. Thus, container experts from Nairobi were involved to share their experiences with containerization technology. They helped determine how access control in containers can be improved. Since they shared the various problems that they face when trying to control access to containers, as it was a big challenge to them.

Subsequently, this research also adopted qualitative research approach since all the data gathered is qualitative in nature. This data will help in identifying and analyzing similar and important themes and patterns from the collected data.

After analyzing the collected data and determining the important and similar themes and patterns, an efficient plugin that makes access decisions based on container users was designed. The main aim of this plugin is to limit the users who can access different containers and defining roles different users can perform within the container environments.

5. RESULTS

The data for this research was obtained from various sources. First there were focus groups discussions with containerization technology experts within Nairobi. Secondly there were numerous tests that were conducted on current container authorization plugins and on the Docker engine to see how it communicates with external plugins. Observations were made from these tests and recorded.

5.1 Data Analysis

The data collected from the focus group was qualitative in nature and it was analyzed using content analysis to identify common patters and themes among the containerization experts.

Table 1 below shows the summarized version of the common patterns and themes identified among the container technology experts involved in the focus group discussion.

Data collected through observation, by performing test on current Docker authorization plugins and on Docker engine. To determine how the engine, communicate with access control plugins was also qualitative. This data was analyzed using framework analysis to identify common themes.

Table 2 below summarizes common themes that were identified on the tests conducted on current container authorization plugins. And also how the container engine communicates and extends its functionalities using third party plugins.

Table 1: Content Analysis for Focus Groups data

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
Concern on access control in docker containers	The main concern is that it is not possible to securely protect applications running within containers. 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 2: Framework Analysis for data from observeations on tests conducted

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.

5.2 Plugin Development

This study led to development of a container access control plugin that will control access in Docker containers based on container users. It has been developed using python Flask framework using HTTP methods to ensure efficient communication with the Docker engine. It uses the. spec way of storing files. This plugin is built only to control which container users can access the containers. All host system users have no access to containers. Also, allowed container users can only access the containers as non-root users, since the plugin also disables the default root access. The Docker access command, exec, has been used to implement the policy file, to make decision on whether a request is allowed or denied access to a container.

5.3 Plugin test results

The individual components developed were tested using Postman API testing tool. The Docker engine communicates with external authorization plugins through three main parts. These parts are defined as routes in API. Firstly, there is '/Plugin.Activate', that is used to test if the plugin works and can communicate with docker engine. Secondly '/AuthZPlugin.AuthZReq', that carries authorization requests from users to the docker engine, which then transfers it to the access control plugin for decision making based on policy. It is also responsible for returning a response to the user from the plugin through the Docker engine based on decision made by the plugin. Lastly the '/AuthZPlugin.AuthZRes' route that defines a valid access response from the plugin.

After testing all the three components in the code and ensuring that they could function as expected. System testing was conducted where all these individual components were integrated and tested to determine if the plugin is performing as expected. System testing was conducted on Linux Operating System running on Ubuntu 18.04 LTS Kernel. Docker engine version 19.03.8 was deployed on the Ubuntu kernel, then Nginx and PostgreSQL docker images were used to create containers and perform the tests. Users were created within the containers and new images were built. The plugin was deployed using a bash script where all dependencies were installed too. The plugin performed as expected. First the default root user access to containers that allows privileged host system users access was disabled by the plugin. Access to containers was based on users within that containers. Not all users within the containers are allowed access, but only those whose access has been allowed on the policy file. Thus, a user cannot access a container he or she is not a member of and he or she must have access rights defined within the policy file.

```
    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 5: Defining Policy file

Figure 5 above shows how the policy is defined within the policy file. User 'titus' is allowed access to all containers within that host using the exec command.

Figure 6 below shows how this study plugin disallows host system users from accessing the containers. Host system user pascal is denied access to containers since he is not an existing user in any container.

```
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 6: Host system users test

```
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 7: Container users Privilege levels

The container users who are allowed access by the policy can only access the containers in unprivileged mode or as non-root users. For a user to perform any change to the contents of the applications, files or directories within the container they must first elevate to privileged. To be able to elevate one's access to privileged mode a user must first have been granted the rights by the container administrator. All users that do not have rights to elevate to privilege mode and can access the containers, can only perform read operations. Figure 7 above shows different privileges that users within a container can have. User 'titus' has sudo rights within the container, thus can elevate the privileges and perform administrative tasks. While, user 'rugendo' does not have sudo rights thus cannot elevate the privileges. This shows that user 'rugendo cannot perform administrative tasks within the container.

To protect the policy file, all host system users were denied privileged access. This is because the policy file is located within the host system. Only users within a specific group for container administrators is allowed privileged access. This means that only members of this group can alter the policy file.

```
#
# 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 8: Denying other host system users privileged access

The default privileged admin and sudo groups have been disabled and group authz given privileged access, this is according to figure 8 above. This means that only users from group authz can modify the policy file. Figure 9 below shows test on different host system users within different groups to determine if they can elevate their privileges and modify the policy file. User 'pascal' who is a member of sudo group and not a member of authz group cannot access the policy file. Whereas user 'elly' who is a member of authz group but not sudo can elevate privileges and access 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 9: Host system users' access to policy file test

The users of authz group can add other container users to the policy file to allow them to access containers. A user added on the policy will only be able to access a container that they already exist in. If the user is not in a container, they will not be able to access that container. Users allowed access on the policy can only access containers that they are a member of. All other users will not be allowed access regardless of whether they are container users or host system users.

After the developed plugin performed as expected. It was shared with several Docker experts to perform usability testing. This was to determine if the plugin is easy to use and if it answers some of the issues that had been raised regarding access control on containers. The response was that the plugin is performing perfectly, and it will help in securing access and what actions users can perform within the containers. It will also simplify auditing since the administrators will focus on specific container logs to determine actions performed by specific container users. The only concern was if this plugin can be integrated with host system users, to make container

access decisions based on them.

6. DISCUSSION

The default access to container systems is based on all or nothing. Where a host system user 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. This research lead 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 access control policy is defined in terms of users created in the container virtual environment rather than host users. 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 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 **[11]**, you either run all commands or denied running any command at all. The authz plugin denies all requests until docker-machine root user or any other host system 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. The resulting access control plugin model denies only access requests but allows all other docker commands to be run by all users. To access a container, the user must exist within the container and must be allowed access into the plugin by the administrator. The plugin also 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. Rather 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. This paper resultant 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. This study 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. For instance, a user in OPA who can run all commands, can also access all containers in root mode. This shows that any privileged host system user can still alter applications and files running within a container. This is because 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. This study resultant 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 and allowed access to the container through the plugin, they gain access as unprivileged users. Thus, a user can only perform actions like write, execute or modification of files if they can elevate to privileged rights. This means that the plugin can allow multiple users to the container and still limit what they can do within the container environment.

The main difference and a downside of this plugin compared to OPA and authz is that this study resultant plugin is concerned with access only. This plugin will not authorize other container commands being run by host system users. This is because the plugin is using container users to make decisions. The major problem with all the plugins is securing the policy file. Since the file in all plugins resides within the host. To achieve this one must ensure that all applications running within that host are running within containers. This is to ensure that only container administrators have privileged access on the host system.

7. CONCLUSION

This research 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. This study was able to evaluate and determine how current container virtualization authorization frameworks implement access control. And, how they interact with container engines. Additionally, the study led to the 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. Also, all host users can also be denied access to containers despite their privileges. The default container root access to containers can be denied using this plugin. This enables the administrator to give different container users different privileges and rights within the containers. Since all allowed users get access in unprivileged mode. Actions to be performed within the container will depend on the privileges given to a specific container user by the administrator.

7.1 Limitations

This study resultant plugin can work best on a host system that is running all applications on containers. If there are some applications running on the host system and having multiple groups with privileged rights, the policy file security might be threatened. This is because the policy file is stored within the host system. Secondly, the developed container access control plugin can only be used to restrict access to docker 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. Finally, this 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.

7.2 Recommendations for Further Work

Currently, the developed plugin can work on a host system that is running all applications on containers. If there are applications running on the host system and having multiple groups with privileged rights, the policy file security might be threatened. This is because the 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.

Docker container developers should also include a way of prompting authorized users for password as they access the containers.

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 host 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.

8. REFERENCES

[1] T. Bui, "Analysis of Docker Security," 2015.

[2] H. Jain, "LXC and LXD: Explaining Linux Containers," 2 June 2016. [Online]. Available: https://www.sumologic.com/blog/lxc-lxd-linux-containers/. [Accessed 27 April 2020].

[3] J. Chelladhurai, P. R. Chelliah and S. A. Kumar, "Securing Docker Containers from Denial of Service," in IEEE International Conference on Services Computing, San Francisco, CA, USA, 2016.

[4] C. Pahl, B. Antonio, J. Soldani and P. Jamshidi, "Cloud Container Technologies: a State-of-the-Art Review," IEEE Transactions on Cloud Computing, p. 1, May 2017.

[5] Z. H. Shoeb and A. Sobhan, "Authentication and Authorization: Security Issues for Institutional Digital Repositories," Library Philosophy and Practice, pp. 1-8, 2010.

[6] F. Hauser, M. Schmidt and M. Menth, "xRAC: Execution and Access Control for Restricted Application Containers on Managed Hosts," ArXiv, vol. abs/1907.03544, pp. 1-9, 2019.

[7] K. Kuusik, "Docker Security – Admin Controls," 19 June 2015. [Online]. Available: https://blog.container-solutions.com/docker-security-admin-controls-2. [Accessed 12 January 2020].

[8] docker Inc, "docker docs," 2019. [Online]. Available: https://docs.docker.com/engine/extend/plugins_authorization/. [Accessed 04 February 2020].

[9] L. Levin, "Docker AuthZ Plugins: Twistlock's Contribution to Docker Security," 18 February 2016. [Online]. Available: https://www.twistlock.com/2016/02/18/docker-authz-plugins/. [Accessed 29 December 2019].

[10] A. Nosek, "Open Policy Agent, Part I - The Introduction," 14 October 2019. [Online]. Available: https://dzone.com/articles/open-policy-agent-part-i-the-introduction. [Accessed 31 December 2019].

[11] E. Toews, "Develop a Docker Authorization Plugin in Python," 30 July 2016. [Online]. Available: https://etoews.github.io/blog/2016/07/30/develop-a-docker-authz-plugin-in-python/. [Accessed 20 February 2020].

[12] D. Lang, H. Jiang, W. Ding and Y. Bai, "Research on Docker Role Access Control Mechanism Based on DRBAC," in Jwenal of Physics: Conference Series, Beijin, 2019.