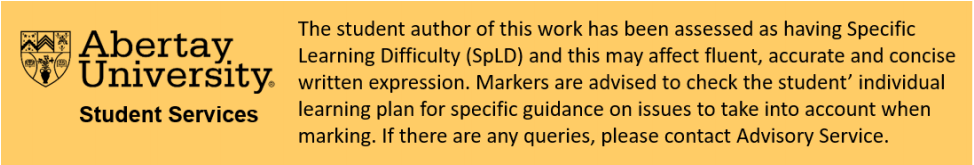
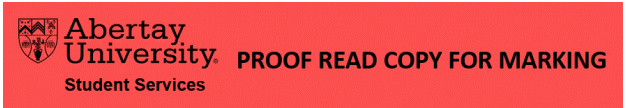


|  |
| --- |
| **Containers viability as a learning tool**  **Aidan Cram**  CMP320: Hacking 3  BSc Ethical Hacking Year 3  2020/21 |





*Note that Information contained in this document is for educational purposes.*

Abstract

This report contains an evaluation of Dockerfiles as a tool for creating authentic learning environments when compared to a virtual machine. The scope of this work will focus on proof of concept and, therefore, there will be a discussion of isolated examples of individual concepts. This will include the main aspects of a machine necessary to configure the creation of an authentic environment. This means that all of the containers will be deliberately streamlined, therefore, the focus will be on the aspect of the container that each is demonstrating as configurable.

To ensure that this is a comprehensive study, four different Dockerfiles were created: one for each main aspect that must be configured. Each was developed and tested independently of each other, with individual performance analysis to improve accuracy. Each Dockerfile has an in-depth explanation outlining their creation. This will ensure that the procedure can be replicated or merged with an existing Dockerfile.

After the creation and performance analysis of these four Dockerfiles, it was clear that the containers granted a lower level of resource usage, while the ability to manipulate the containers was as easy as it would be with a virtual machine. Potentially, the ability to easily randomize aspects of the container could possibly make it a superior method for creating authentic learning environments.

+Contents

[1 Introduction 1](#_Toc72270803)

[1.1 Background 1](#_Toc72270804)

[1.2 Aim 2](#_Toc72270805)

[2 Procedure 3](#_Toc72270806)

[2.1 Dockerfile Implementation 3](#_Toc72270807)

[Telnet 4](#_Toc72270808)

[Privilege Escalation 5](#_Toc72270809)

[Vulnerable Service 6](#_Toc72270810)

[Network 8](#_Toc72270811)

[2.2 Generating Dockerfiles 9](#_Toc72270812)

[2.3 Performance analysis 10](#_Toc72270813)

[3 Discussion 13](#_Toc72270814)

[3.1 General Discussion 13](#_Toc72270815)

[3.2 Future Work 14](#_Toc72270816)

[4 References 15](#_Toc72270817)

[5 Bibliography 16](#_Toc72270818)

[6 Appendices 17](#_Toc72270819)

[Appendix A – Dockerfiles 17](#_Toc72270820)

[Appendix B – Python 19](#_Toc72270821)

# Introduction

## Background

Traditional teaching methods usually involve performing standard exercises from a textbook. However, teaching in cyber security benefits from having authentic real-world examples. According to Hartley, students “need to be equipped with skill sets currently used by attackers” (D. Hartley, 2021). This demonstrates that there is more to hacking than mearly understanding how the systems work. The fact that students need to understand the practical aspects of hacking (as well as the concepts of computing) is further supported by Medlin and Houlik who state, “Technical expertise is necessary to implement the details of a security operation” (Hartley, Medlin and Houlik, 2021).

This is, in part, because no digital system or program exits within a vacuum, as there will rarely be two situations in cyber security that are identical and there will rarely be a real-world example identical to a textbook scenario. This is because computer systems are hugely complex and are interconnected with many programs running simultaneously. Teaching students about cyber security without resorting to repeated use of a textbook is important in preparing students for real-world situations: this is where virtualisation can be applied.

With virtualisation, it is possible to simulate an approximate representation of what a student may expect to encounter after leaving formal education. They could be presented with a machine that has all the characteristics and complexities they would find in any real situation. The important point is, however, that these machines can be manipulated and tailored to be vulnerable to specific types of attacks. Also, these machines can be randomised to ensure that students are not faced with the same situation twice.

There are two main types of virtualisation which can be utilised for teaching purposes: Virtual Machines (VMs) and Containers. Both involve the simulation of a virtual environment and both can be configured to provide different scenarios to students. However, they differ mainly in the scope of their implementation.

A VM creates a complete genuine copy of the operating system (OS) it is set up with and will act in the same way as a computer would with that OS in any given situation. It can be preloaded with applications and files, configured to the correct specifications and then distributed to students who can then run it on software such as VMware or VirtualBox. This provides an authentic experience for the learner as it gives them the freedom to tackle the task in different ways. This allows the learner to expand upon the work laid out by the lecturer which will, in turn, encourage more independent learning. The main constraint with VMs is that they must share hardware resources with the host machine. This can make it difficult for students to access these VMs on less powerful computers; a problem which grows exponentially when the concept of a virtual network is considered, where the student must run multiple VMs simultaneously.

A Container, by comparison, operates as a partition of the host machine and shares its resources. This means that a Container must run on the OS and architecture of the host machine, however it does not use up additional resources for running the kernel and guest OS. This leads to greatly reduced overheads, thereby making it easier to run multiple containers on the same system. However, the reduced scope of the container does not represent an authentic system where learners are free to explore different methods of completing the task.

The benefits of containers can best be seen in systems where multiple containers are running simultaneously. In a system where more machines are being run as VMs, the amount of required resources can quickly exceed those available on the machine.

## Aim

The aim of this project is to ascertain and then demonstrate Dockerfiles’ ability to configure containers in order to create authentic leaning environments. To this end, it was decided that it was important to examine whether if the necessary components of a machine could be successfully configured using Dockerfile. These components were determined to be:

* Services on ports
* Non-root login status
* Installable services
* Networking capabilities

A Dockerfile will be created to configure each of these aspects of the container in order to determine whether containers are as configurable as a virtual machine.

The next step was a performance analysis of the containers and a comparable virtual machine to determine what, if any, performance benefits can be obtained by using a container instead of a virtual machine. This analysis will focus on the memory and CPU usage of these systems, as these are the most quantifiable comparative metrics.

# Procedure

## Dockerfile Implementation

The procedure for this project will involve several steps. The first step is the creation of custom containers using Dockerfile. These will be proof of concept of how a container can be configured with vulnerabilities. In order to create these authentic environments, the process was broken down into four stages. Each stage represents an example of a critical aspect of the container that needs to be configured. These stages should also demonstrate that vulnerabilities can be injected into the container.

There are many ways of setting up a Container, however, for the purposes of this report, Dockers Dockerfile feature was used. A Dockerfile contains configurations and commands that allow for the creation of a custom Container built to the given specifications. These configurations can include anything from the operating system being emulated, to installing software and running any command that can be run on a normal system. This is achieved using a number of different commands. First is the **FROM** command as this allows for the identification of the operating system you want the Container to use. For example:

**FROM ubuntu**

The **RUN** command allows for the installation of software onto the Container. It may be important to note that these commands will run when the image is created, and not when the Container is started. These commands look like the following:

**RUN apt-get install -y build-essential**

The last of the essential commands is the **COPY** command. This takes a file on the host machine and copies it to the location given in the command. For example:

**COPY File Destination**

Once the Dockerfile has been configured, it is used to craft an image file. This image will contain all the special configurations that have been specified in the Dockerfile. The image functions as a template that the Containers can be built from. Once an image has been created using the “build” command, the Container can be created using the “run” command, as seen below:

**docker build -t project**

**docker run -dit project**

Then, using the “**docker ps**” command, it is possible to see a list of running Containers.



Figure - ps

One important step when making the Containers was to label each Container. Doing this streamlines the process as it means that it is not necessary to check the name of the Container if it needs to be identified for the “connect” or “exec” commands. However, since docker, understandably, does not allow for multiple containers with the same name, it is therefore necessary to stop and remove the Container after use. It was found that the most efficient way of doing this was via a combination of two commands. The first command stopped all active Containers while the second command identified all containers tagged as “exited”, thereby meaning they were no longer running and where removed from the list.

**docker stop $(docker ps -q)**

**docker rm $(docker ps --filter "status=exited" -q)**

### Telnet

One of the easiest vulnerabilities to test on a Container is via having a vulnerable service running on an open port. To demonstrate this, the Telnet service will be used as it is well-known for being vulnerable and is easy to install. The first part of this two-part process can be achieved by installing Telnet. This can be achieved using the RUN command (as shown earlier) and identifying the Telnet package to be installed.

**RUN apt-get install telnetd -y**

|  |  |  |  |
| --- | --- | --- | --- |
| The second part of this process is to open port 23 using the **EXPOSE** command. This will designate the port on which this should be exposed. What this means, when implemented, is that the first number, **number** **A** in Figure 2 - Expose command, is the port that will be opened on the Docker address while the second, **number B** in Figure 2 - Expose command, is the port that will be opened on the Container. An example of the resulting file can be found in Appendix A **-** Dockerfile Telnet**.** |  | **A** | **B** |
| **EXPOSE** | **23** | **23** |
| Figure - Expose command | | |

The commands needed to run the Container are as follows:

**docker build -t telnet .**

**docker run -dit telnet --name Telnet**

The last part of this implementation is to assign the Container an IP address which can then be targeted.

**docker network connect –ip 192.168.0.15 net Telnet**

|  |  |
| --- | --- |
| To check if this has worked, it is possible to connect to the machine using the Telnet command as shown in Figure 3 - telnet command | Figure - telnet command |

### Privilege Escalation

One direct example of how Containers can be used for education is via their ability to allow for privilege escalation. This is a simple set up as it only requires selecting a starting user that does not have root privileges. This can be achieved by running a command that will add a new user similar to one run in a normal Ubuntu machine.

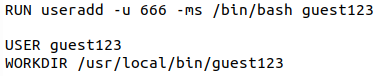


Figure - Dockerfile useradd command

It is possible to verify that this works by building and running the Container then connecting it into the machine. All of this can be achived with the two standard commands:

**docker build -t usersssss**

**docker run -dit user --name user**

**docker exec -it user bash**

It is then possible to use the **cat** command to look at the contents of the /etc/passwd file. Here it can be seen that a user has been created with the name “guest123” (Oosterhof, 2021)

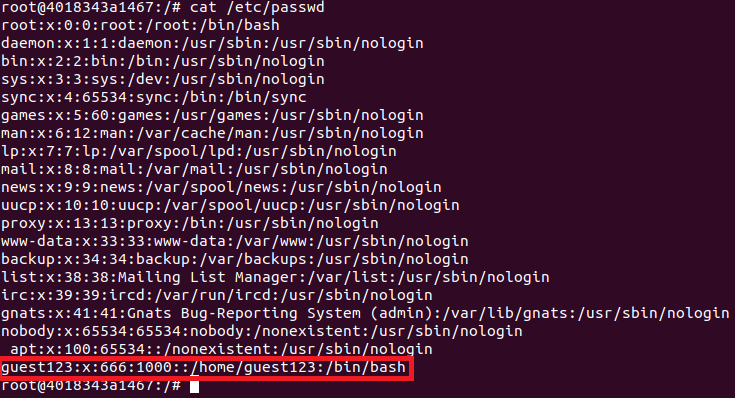


Figure - content of passwd file

The next step is to use the Docker **USER** command. This can be used to specify what user should be used when executing all following commands and where the last user specified will be the one logged onto the Container by default when the Container runs.

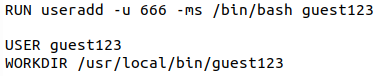


Figure - Dockerfile set user and directory command

This means that when the Container has been rebuilt and run with these lines, it will start as the user specified. This can be seen in Figure 7 - example login to custom account.

****

Figure - example login to custom account

### Vulnerable Service

A second example consists of installing a vulnerable program on the Container. This is an appropriate example to use because it shows how when wanting to demonstrate a vulnerability on a single program, there are lots of unnecessary processes involved with running only that specific program. For this example, the Damn Vulnerable Web Application (DVWA) will be used. This will involve the installation of a number of individual packages, as shown below:

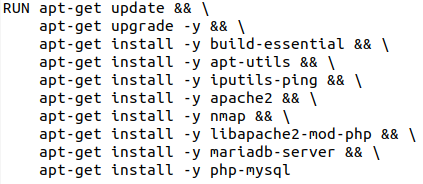


Figure - list of dependancy installs

After the software that the web application requires has been installed, the next step is to copy the DVWA files and other configuration files onto the Container. This can be achieved by using the Dockerfiles **COPY** command. This takes in two arguments; the first is the file being copied and the second is the location to copy it to.



Figure - list of copied files

Once the Dockerfile has been prepared, the Container must be built and run. In this instance, it will again be necessary to assign an IP address so that there is an easily accessible address to connect to for the web application. Doing this also eliminates the chances of a clash with the ports if the docker address is already in use.

**docker build -t service .**

**docker run --name vulnerable\_service -dit service**

**docker network connect --ip 192.168.0.10 net vulnerable\_service**

To verify that all of this works as intended, it is possible to connect to the Container using the “exec” command before using the service command to ensure that everything works as intended.

**docker exec -it testing bash**

**service –status-all**

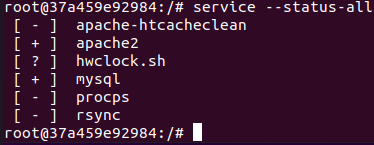


Figure - running services

Looking at “apache2” and “mysql”, the [+] next to them denotes that they are successfully running on the container.

Once this is done, it is possible to connect to the service by “192.168.0.10/index.php”; this will then redirect to “setup.php” and prompt the setup of the system which only requires the “Create/Reset Database” to be clicked.

|  |  |
| --- | --- |
| Doing this will ensure that any database files that are missing will be generated which will then redirect the user to the “login.php” page. | Figure - Database Setup |
| The default login for this has been set to:  Username: admin  Password : password  However, this can be configured in the config settings if so desired. | Figure - Login Screen |
| This will then redirect the user to the DVWA main page where it will now be possible to access all the material it offers. | Figure - DVWA Welcome Page |

### Network

One area where the benefits of using Containers can best be seen is when multiple machines need to be simulated. The most simplistic example of this is an exercise showing a firewall set up between two Containers. To demonstrate this, it is useful to build on the Telnet example from earlier. It is important to note that the default firewall on the Containers will ensure that any port, even ones opened using EXPOSE, will be closed if there is no service running on them. Using this template, it is possible to write a new Dockerfile as seen in Appendix A **-** Dockerfile network. For this example, it is important as part of set up the network to assign two Containers IP addresses on this network. Doing this allows for communication between the Containers which will be shown using the Telnet service.

The first step is the setup; this involves creating the subnet using the “**network create**” command.

**docker network create --driver=bridge –subnet=192.168.0.1/16 net**

This will create a network according to the specifications in the command. It is important to note the name assigned to the network (**net** in this case), as this will be used to connect Containers to this subnet. It is possible to check the creation of this network using the “network is” command and then checking the configuration using the “network inspect” command.

Once the network has been successfully setup, the next step is to build and run the Containers.

**docker build -t network .**

**docker run --name Net1 -dit network**

**docker run --name Net2 -dit network**

The last stage of the setup is to assign the Containers to the subnet. This is done using another of the series of “network” commands. In this case, it is the “network connect” command. This takes the form of three arguments: the IP address to assign, the name of the network and the name of the Container.

**docker network connect --ip 192.168.0.10 net Net1**

**docker network connect --ip 192.168.0.11 net Net2**

Now that these are set up, the next step is to connect to one of the Containers.

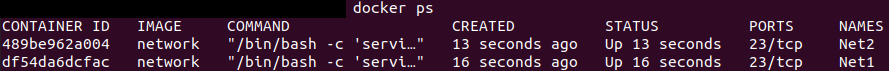
****

Figure - docker ps command

This can be done using the docker **exec** command. This is the equivalent of logging into the machine as root which will subsequently give access to the command line. Below is an example for connecting to the Net1 Container.

**docker exec -it Net1 bash**

Next, it is important to verify that the port has opened properly. This can be done using the nmap command, as shown below:

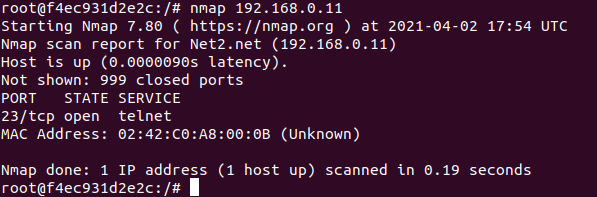


Figure - nmap results

Now that it has been verified as having been setup correctly, it is possible, from this Container, to use the Telnet service to connect to the Net2 Container. This can be achieved using the Telnet command, as seen below.

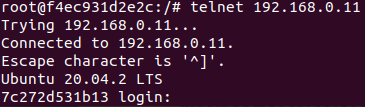


Figure - telnet login

## Generating Dockerfiles

for a practical exercise, it can be desirable to have aspects randomized in some way. This means a student can try the same approach on a slightly differently configured machine to ensure that they understand the techniques or to ensure that, if students are assisting each other, they are explaining techniques and not simply passing along the answers.

This is easily achievable within Dockerfiles, as it is possible to generate these with a python script. To prove the success of this, the Privilege Escalation script was generated with the python script, as seen in Appendix B – Python. The premise here is to write a script that will primarily write the text from a successful Dockerfile into a new text document.

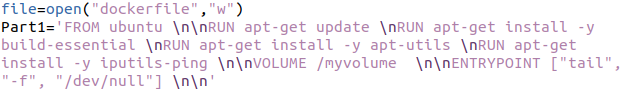


Figure - Python creating docker RUN commands

|  |  |
| --- | --- |
| However, it is then possible to take this a step further by using pythons’ random function to randomize an aspect of the file. In this case, it is the name of the user being generated.  The random variable can then be used in the generation of the Dockerfile. | Figure - user randomiser code |
| Figure - Python creating Docker USER command | |
| This results in the user created by the Dockerfile being randomly assigned by the python script. |  |
| Figure - example USER commands |

Now that the Dockerfile has been created, an example of both potential examples can be seen in Appendix A - Generated Dockerfile; these can be run to test that it works as expected. This uses the same commands as previously demonstrated, a further example of which can be seen below.

**sudo docker build -t gen .**

**sudo docker run --name generated -dit gen**

**sudo docker exec -it generated bash**

By doing this with both different versions of the generated file, it is possible to see that these successfully create working Dockerfiles and prove that it is possible to generate Dockerfiles with random characteristics.





Figure - User loggins

## Performance analysis

An important aspect of the analysis of Containers is to see how these compare in resource consumption with virtual machines. When considering these resources, the ones to concentrate onthe CPU and memory usage.

To do this on the Containers, the docker “status” command can be used to monitor resource usage on each individual Container.

|  |
| --- |
|  |
|  |
|  |
|  |

Figure - container performance results

This can then be compared to the resource consumption of a virtual machine. To make this as reasonable a comparison as possible, Task Manager will be used to monitor the resource usage of VMware when running only a basic installation of Ubuntu.



Figure - Virtual Machine performance results

It is important to note that Task Manager only records CPU usage in 0.1% increments, while docker does so in 0.01% increments. Likewise, with memory, Task Manager uses 0.1 Megabytes as its smallest value, while docker uses 0.001 Mebibytes. This means that accuracy and comparison can be more difficult to obtain.

Figure - Container memory results

The average memory usage of the docker Containers is 4.96 MiB, which is equivalent to 5.45 MB. This can then be compared to the 47.1 MB used by VMware. The next step is to look at the CPU usage. This is simpler to obtain as it has been measured on the same scale for both the Container and the VM, but with different levels of sensitivity.

Figure - Memory comparison

The difference in CPU usage is almost indistinguishable. The VM used >0.1% while the Container used at most 0.01%. This means that the VM could be using up to an order of magnitude more, however regardless of this, both of these numbers are so small that the difference is very minor.

# Discussion

## General Discussion

Looking at these different Containers, it is possible to see how they could be used in the creation of a configured (yet authentic) learning environment. For example, opening the Telnet port shows how ports and services can be configured in the Container. This is then expanded upon with the implementation of DVWA as this shows how more complex services and programs, with lots of dependencies and component parts, can be implemented. The network Container also expands on the ability of the Containers to connect between each other and set up a virtual network. These are obviously very simple examples intended as proof of concept, however, there is no reason that these could not be combined to design increasingly complex Dockerfiles.

To this end, it is important to consider the wider uses of the ability to generate Dockerfiles. While the proof of concept generator was simplistic, it is possible to use this to, for example, randomize what ports are opened with services running on them. This could then be combined with the techniques shown in the Network Dockerfile to create a whole, randomly generated, network of machines that a student would be presented with.

While all of these Dockerfiles succeed in achieving what they were individually designed to, while proving that Dockerfile can be used to create authentic environments that are completely configurable, the key benefit of Containers over VMs is, supposedly, their reduced resource consumption. Therefore, the results of the Performance Analysis stage cannot be overlooked.

The result of this stage was conclusively that Containers could use as little as 1% of the amount of memory that a full VM would use, although the average was closer to 10%. Either way, this shows a remarkable reduction in memory usage when using Containers. This would only become more pronounced as networking is considered and where multiple Containers could be used instead of VMs. It should be noted, however, that more complex and realistic Containers are more likely to resemble the memory usage of the vulnerable service container which had DVWA running on it. It is, therefore, not an unreasonable extrapolation to estimate an average of 20MB used for a Container running DVWA and a few other features. This would, however, still result in a memory usage being less than 50% of a VM, thereby showing that, even at its most complex, a Container can provide significant resource benefits.

The other aspect of performance is CPU usage; something which is more difficult to analyse. This is because the numbers are so small that it is difficult to conclusively state that either is better, or even that they are comparable. This is due to the limitations of Task Manager to provide a sufficient degree of accuracy as CPU usage is only measured in 0.1% increments. This means that the 0% used by the VM actually means >0.1%, which could still be significantly more than the 0.01% recorded by the Container. Therefore, this report can only state that the Container may use less CPU resources than its VM counterpart, however no definitive evidence can prove this either one way or the other.

From the above, it can be concluded that Containers, at least using Dockerfile, have both sufficient configurability and resource benefits to not only provide learning environments that are as authentic as VMs, but do so with largely reduced resource usage.

## Future Work

In future, terraform could be used to orchestrate multiple Containers to create a system of configured Containers maintained using a stable and fast platform.

More configuration could also be considered, however those in this report cover most proof of concept configurations and anymore configurations would likely only be combining the principles covered already. Despite this, the expanded potential of the generation python file provides multiple new angles from which to improve the authenticity of the Containers.

# References

D. Hartley, R., 2021. [online] Scholarworks.lib.csusb.edu. Available at: <https://scholarworks.lib.csusb.edu/cgi/viewcontent.cgi?referer=https://scholar.google.com/&httpsredir=1&article=1055&context=jitim> [Accessed 18 May 2021].

Hartley, R., Medlin, D. and Houlik, Z., 2021. [online] Proc.iscap.info. Available at: <http://proc.iscap.info/2017/pdf/4341.pdf> [Accessed 18 May 2021].

Oosterhof, N., 2021. *How to create new users in a Docker container?*. [online] net2. Available at: <https://net2.com/how-to-create-new-users-in-docker-container/> [Accessed 17 May 2021].

# Bibliography

Blaha, M., 2021. *Dockerfile - How to copy files from a local folder?*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/64410123/Dockerfile-how-to-copy-files-from-a-local-folder> [Accessed 19 May 2021].

MariaDB KnowledgeBase. 2021. *Creating a Custom Docker Image*. [online] Available at: <https://mariadb.com/kb/en/creating-a-custom-docker-image/> [Accessed 19 May 2021].

Riptutorial.com. 2021. *Docker - Monitoring resource usage | docker Tutorial*. [online] Available at: <https://riptutorial.com/docker/example/5560/monitoring-resource-usage> [Accessed 19 May 2021].

Braincoke.fr. 2021. *DVWA - Start with Docker - Braincoke | Security Blog*. [online] Available at: <https://braincoke.fr/write-up/dvwa/dvwa-start-with-docker/> [Accessed 17 May 2021].

sneppets. 2021. *How to add new user to the Docker container using Dockerfile -*. [online] Available at: <https://www.sneppets.com/cloud/docker/how-to-add-new-user-to-the-docker-container-using-Dockerfile/> [Accessed 19 May 2021].

HowtoForge. 2021. *How to Install and Use Telnet on Ubuntu 20.04 LTS*. [online] Available at: <https://www.howtoforge.com/how-to-install-and-use-telnet-on-ubuntu/> [Accessed 17 May 2021].

Howe, J., 2021. *VMware Provisioning using Hashicorp Terraform – VirtJunkie*. [online] Virtjunkie.com. Available at: <https://www.virtjunkie.com/vmware-provisioning-using-hashicorp-terraform/> [Accessed 17 May 2021].

Thomaslaurenson.com. 2021. *Installing and Configuring DVWA*. [online] Available at: <https://www.thomaslaurenson.com/blog/2018-07-12/installing-and-configuring-DVWA/> [Accessed 17 May 2021].

Modi, S., Sabadra, P., KAWADIWALE, G. and Sabadra, P., 2021. *Dockerfile Reference – RUN & CMD Instructions | Knowledge Junction*. [online] Knowledge Junction. Available at: <https://knowledge-junction.com/2021/02/10/Dockerfile-reference-run-cmd-instructions/> [Accessed 19 May 2021].

Onsem, W., 2021. *How to use the Random Function in a if statement, in Python 3?*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/48045444/how-to-use-the-random-function-in-a-if-statement-in-python-3> [Accessed 17 May 2021].

GitHub. 2021. *opsxcq/docker-vulnerable-dvwa*. [online] Available at: <https://github.com/opsxcq/docker-vulnerable-dvwa> [Accessed 17 May 2021].

Singh, S., 2021. *Dockerfile | Guide to How Dockerfile works in Docker? with Advantages*. [online] EDUCBA. Available at: <https://www.educba.com/Dockerfile/> [Accessed 19 May 2021].

Wallen, J., 2021. *Docker Basics: How to Use Dockerfiles - The New Stack*. [online] The New Stack. Available at: <https://thenewstack.io/docker-basics-how-to-use-Dockerfiles/> [Accessed 19 May 2021].

# Appendices

## Appendix A – Dockerfiles

#### Dockerfile Telnet

#Telnet

FROM ubuntu

RUN apt-get update

RUN apt-get install -y build-essential

RUN apt-get install -y apt-utils

RUN apt-get install telnetd -y

EXPOSE 23:23

VOLUME /myvolume

ENTRYPOINT ["/bin/bash","-c","service openbsd-inetd start && tail -f /dev/null"]

#### Dockerfile User

#This is a sample Image

FROM ubuntu

RUN apt-get update && \

apt-get install -y build-essential && \

apt-get install -y apt-utils && \

apt-get install -y iputils-ping && \

VOLUME /myvolume

ENTRYPOINT ["tail", "-f", "/dev/null"]

RUN useradd -u 666 -ms /bin/bash guest123

USER guest123

WORKDIR /usr/local/bin/guest123

#### Dockerfile Vulnerable Service

#This is a sample Image

FROM ubuntu

MAINTAINER "opsxcq@strm.sh"

ENV TZ=Europe/Kiev

RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone

RUN apt update

RUN apt install -y python3-pip

EXPOSE 1234:80

RUN apt-get update && \

apt-get upgrade -y && \

apt-get install -y build-essential && \

apt-get install -y apt-utils && \

apt-get install -y iputils-ping && \

apt-get install -y apache2 && \

apt-get install -y nmap && \

apt-get install -y libapache2-mod-php && \

apt-get install -y mariadb-server && \

apt-get install -y php-mysql

RUN echo "ServerName 172.17.0.2" >> /etc/apache2/apache2.conf

RUN mkdir -p /var/www/html

COPY DVWA-master/ /var/www/html/

COPY php.ini /etc/php5/apache2/php.ini

COPY config.inc.php /var/www/html/config/

RUN chown www-data:www-data -R /var/www/html && \

rm /var/www/html/index.html

RUN service apache2 start && \

service mysql start && \

sleep 3 && \

mysql -uroot -pvulnerables -e "CREATE USER app@localhost IDENTIFIED BY 'vulnerables';CREATE DATABASE dvwa;GRANT ALL privileges ON dvwa.\* TO 'app'@localhost;"

ENTRYPOINT ["/usr/sbin/apache2ctl", "-DFOREGROUND"]

#### Dockerfile network

FROM ubuntu

RUN apt-get update

RUN apt-get install -y build-essential

RUN apt-get install -y apt-utils

RUN apt-get install -y iputils-ping

RUN apt-get install -y nmap

RUN apt-get install telnetd -y

RUN apt-get install telnet -y

EXPOSE 23:23

VOLUME /myvolume

ENTRYPOINT ["/bin/bash","-c","service openbsd-inetd start && tail -f /dev/null"]

#### Generated Dockerfile

FROM ubuntu

RUN apt-get update

RUN apt-get install -y build-essential

RUN apt-get install -y apt-utils

RUN apt-get install -y iputils-ping

VOLUME /myvolume

ENTRYPOINT ["tail", "-f", "/dev/null"]

RUN useradd -u 666 -ms /bin/bash NeverGive

USER NeverGive

WORKDIR /usr/local/bin/NeverGive

## Appendix B – Python

import random

file=open("Dockerfile","w")

Part1='FROM ubuntu \n\nRUN apt-get update \nRUN apt-get install -y build-essential \nRUN apt-get install -y apt-utils \nRUN apt-get install -y iputils-ping \n\nVOLUME /myvolume \n\nENTRYPOINT ["tail", "-f", "/dev/null"] \n\n'

Name=0

rand = random.randint(1,2)

if rand==1:

Name="NeverGive"

else :

Name="YouUp"

Part2="RUN useradd -u 666 -ms /bin/bash " + Name + "\n\nUSER " + Name + " \nWORKDIR /usr/local/bin/" + Name

file.write(Part1+Part2)

file.close()