lyit | Institiúid Teicneolaíochta Leitir Ceanainn | Letterkenny Institute of Technology

Computing Department, Letterkenny Institute of Technology, Port Road, Letterkenny, Co. Donegal, Ireland.

An Assessment of Containerised Infrastructure in Meeting PCI-DSS Requirements

Author: Dane Coyle

Supervised by: Ruth G. Lennon

A thesis submitted in partial fulfilment of the requirements for the

Bachelors in Science in Computing in Data Centre Management

Submitted to Quality and Qualifications Ireland (QQI)
Dearbhú Cáilíochta agus Cáilíochtaí Éireann

July 2020

Declaration

I hereby certify that the material, which l now submit for assessment on the programmes of study leading to the award of Bachelors of Science in Computing in Data Centre Management, is entirely my own work and has not been taken from the work of others except to the extent that such work has been cited and acknowledged within the text of my own work. No portion of the work contained in this thesis has been submitted in support of an application for another degree or qualification to this or any other institution. I understand that it is my responsibility to ensure that I have adhered to LYIT's rules and regulations.

I hereby certify that the material on which I have relied on for the purpose of my assessment is not deemed as personal data under the GDPR Regulations. Personal data is any data from living people that can be identified. Any personal data used for the purpose of my assessment has been pseudonymised and the data set and identifiers are not held by LYIT. Alternatively, personal data has been anonymised in line with the Data Protection Commissioners Guidelines on Anonymisation.

I give consent for my work to be held for the purposes of education assistance to future Computing students at LYIT and it will not be shared outside the Computing Department of LYIT. I understand that my assessment may be shared with any other third party and will be held securely in LYIT in line with the Records Retention Policy.

Signature of Candidate:  Date: 11/07/2020

Abstract

Containers as a form of virtualisation are becoming increasingly popular, and their use in industry is an enticing replacement for traditional virtual machines in terms of cost and performance. When considering the adoption of a new technology, it must be assessed against relevant compliance standards, in this case PCI-DSS. This report found that while the underlying suitability of the technology was viable, the actual testing and implementation proved problematic. There is enough success in meeting high-level PCI-DSS requirements to justify investigating this ambiguity, and seeing what compromises and concessions have to be made in order to deploy a PCI-DSS compliant infrastructure.

Acronyms

Acronym	Definition	Page
PCI-DSS	Payment Card Industry Data Security Standard	1
VM	Virtual Machine	1
PCI-SSC	Payment Card Industry Security Standards Council	2
IaC	Infrastructure as Code	2
SDP	Site Data Protection	5
CISP	Cardholder Information Security Program	5
SDN	Software-Defined Networking	10
DSC	Desired State Configuration	12
NAT	Network Address Translation	20
PIE	Position Independent Executables	24

Table of Contents

Table of Figures

Table of Tables

Table of Code Listings

1. Introduction

The main aim of this project was to assess the viability of a containerised infrastructure in meeting current Payment Card Industry Data Security Standard (PCI-DSS) requirements. An additional aim was to do so using automated scripts capable of consistently reproducing the desired outcomes. The central conceit in this work is that only one compliance standard is of concern, an actual business subject to PCI-DSS would likely also be subject to GDPR and other information security standards. Given the exhaustive nature of the individual PCI-DSS requirements, this project has limited the assessment criteria to meeting only the twelve high-level requirements stated in the latest PCI-DSS documentation.

1.1. Purpose

The purpose of this project is to research PCI-DSS, and assess the viability of a containerised infrastructure in fulfilling its requirements. This document records the objectives, implementation, and results of this project. It serves as a complement to other artefacts hosted externally on GitHub, as evidence the described work has been carried out. All referenced material and external links to necessary files have been included in this document.

1.2. Background

The adoption of containerisation in business has continued to increase. While established virtual machine (VM) usage – that is to say a dedicated VM running one or more applications or services – is unlikely to be completely supplanted, containers present an alternative with a multitude of benefits. More containers can be run on a machine and use less resources than an equivalent VM setup, all while providing better performance (Seo et al, 2014). In addition containers integrate well with modern DevOps processes, offering unprecedented agility in development and running applications, especially in a cloud environment (Kang et al, 2016). These reduced resource requirements and increased efficiencies could present major cost savings for businesses, in addition to reduced turnaround on development times.

Container usage is likely to continue increasing, and in doing so will influence relevant compliance standards. Until recently these standards have primarily focused on requirements around traditional virtual machines. There can still be issues implementing new requirements using containerisation, due to the relatively new addition of such requirements to existing standards. PCI-DSS, the standard issued by the Payment Card Industry Security Standards Council (PCI-SSC), deals with the storage of credit card information. Section E7, which specifically addresses containers, was only added in 2018 as supplemental material (PCI-SSC, 2018a). The risk associated with a data breach or non-compliance to a business with this level of sensitive information could be severe, and ambiguity and oversight in updated standards documentation may make container compliance difficult to achieve.

Virtualisation in general has advanced to point where whole infrastructures can be implemented entirely though code as containers. Implementing infrastructure as code (IaC) in a secure and compliant manner is a necessity. Secure programming and networking principles are a consideration from the start of any project, not as a later addition. Automated scripts are able to deploy these infrastructures at massive scales, so making these scripts secure and their results compliant is an essential process.

1.3. Problem Statement

Containers have only recently been added to PCI-DSS requirements. As a less proven alternative to virtual machines, this leads to questions about their viability as an option for businesses wishing to be PCI-DSS compliant. Considering their benefits over virtual machines, a lack of container usage due to compliance concerns could impact a business's overall performance and ability to grow.

1.4. Research Question

The research question considered in this research is:

> How viable is a containerised infrastructure, deployed via automated scripts, as an option for meeting current PCI-DSS requirements?

2. Literature Survey

This literature survey attempts to specifically address the component areas of the research question. The areas of interest and technology explored were those directly relating to the problem statement. These areas have background information included with relevant issues and applications to this project. The scope of the literature reviewed was limited to academic and industry publications.

By breaking down the research question and overall aims of this project, the areas to be surveyed were determined to be as follows:

- PCI-DSS and its implementation requirements.
- Containers and their vulnerabilities.
- Virtualised infrastructure with a focus on software-defined networking.
- Automation of infrastructure as code and the orchestration of automated processes.

2.1. Problem Context

The focus of this review is PCI-DSS and its requirements in the context of industry application. From here the technologies within the scope of this project are explored, narrowing in on the specific aspects that make them suitable for use in applying PCI-DSS. In addition, potential issues and vulnerabilities of these technologies are explored. Finally, the development and automation of these technologies in a PCI-DSS compliant manner will be addressed.

2.2. PCI-DSS

Before the creation of the PCI-SSC, credit card providers including Visa and MasterCard each had their own set of data security standards they required retailers and financial institutions to uphold. Compliance with multiple standards was a significant burden for businesses (Wright, 2009). In 2004 VISA and MasterCard established the first version of the PCI-DSS, a set of standards comprised of elements of VISA's Cardholder Information Security Program (CISP) and MasterCards' Site Data Protection (SDP). PCI-DSS continued to incorporate requirements from other card providers, and in 2006 the PCI-SSC was formed. PCI-DSS continues to be updated with the latest version, 3.2.1, being published in May 2018. As of March 2019, work has begun on version 4.0.

A 2009 hearing before the United States House of Representatives investigated the effectiveness of PCI-DSS compliance in reducing cybercrime. Their investigation found that even full compliance with PCI-DSS does not guarantee the prevention of data breaches (US GPO, 2010). The report noted the outdated nature of the credit card technology of the time, citing the effectiveness of the UK's Payments Association in reducing credit card fraud by introducing chip-and-pin technology. This demonstrates how even effective compliance standards can be hampered by out-dated technology. While the technology used improved and the standards updated to account for them, breaches continued to occur. A 2018 journal addressed this, investigating increasing security breaches of credit card data even among PCI-DSS compliant businesses (Moldes, 2018). The journal stated that only 29% of businesses were still PCI-DSS compliant one year after this status was awarded to them. The problem seems to be with the implementation of PCI-DSS by these businesses, and their failure to regularly check for non-compliance. A 2015 Verizon report stated the following:

> "We have found that of all the data breaches our forensics team has investigated over the last 10 years, not a single organization was PCI DSS compliant at the time of the breach." (Verizon, 2015).

This correlation between security breaches and non-compliance was corroborated in a later 2019 report, also by Verizon. In this they were able to break down by what degree

businesses failed in their compliance, giving specific requirements failures and comparisons to a business's previously recorded compliance levels. The full report goes into great detail, but the following points are particularly relevant to the scope of this project (Verizon, 2019):

- No organisation that was breached was in compliance with all PCI-DSS requirements.
- Compared with other industries, IT services only accounted for 2.7% of all card data breaches over a six year period.
- 26.5% of breaches were directly caused by failing to develop and maintain secure systems.
- 18.4% of breaches were directly caused by failing to maintain a proper firewall configuration.
- 12.2% of breaches were caused by failing to implement authentication as part of access to data.

From these statistics, it is clear that the issue most businesses have is implementing secure infrastructure. Judging by the numerous reports and investigations over the years, it would appear that PCI-DSS is an effective set of requirements in preventing data breaches. Problems arise in its implementation, either through oversight or lack of continual review. As 57.1% of breaches were caused directly by system, authentication and development issues, this is something the aims of this project can be tested against. If a containerised infrastructure can successfully meet these three requirements, it has already been secured against the most prevalent causes of breaches.

2.2.1. PCI-DSS requirements

The PCI-SSC provides a detailed requirements guide for the core PCI-DSS principles. These principles consist of twelve requirements that can be divided into six categories. These principles come with associated testing procedures, which are explored in a later section of this report. As stated in the latest PCI-DSS document, these are as follows (PCI-SSC, 2018b):

Table 1: PCI-DSS High-Level Requirements Overview

Category	Specific Requirement
Build and Maintain a Secure Network and Systems	1. Install and maintain a firewall configuration to protect cardholder data.
	2. Do not use vendor-supplied defaults for system passwords and other security parameters.
Protect Cardholder Data	3. Protect stored cardholder data.
	4. Encrypt transmission of cardholder data across open, public networks.
Maintain a Vulnerability Management Program	5. Protect all systems against malware and regularly update anti-virus software or programs.
	6. Develop and maintain secure systems and applications.
Implement Strong Access Control Measures	7. Restrict access to cardholder data by business need to know.
	8. Identify and authenticate access to system components.
	9. Restrict physical access to cardholder data.
Regularly Monitor and Test Networks	10. Track and monitor all access to network resources and cardholder data.
	11. Regularly test security systems and processes.
Maintain an Information Security Policy	12. Maintain a policy that addresses information security for all personnel.

As stated in the introductory section, the scope of the project has been limited to these twelve high level requirements. These requirements have been expanded in the implementation and testing sections of this document.

2.3. Containers

Containerisation is a form of virtualisation that enables operating system consolidation at the hypervisor level, allowing independent instances of the same application to run concurrently on the same hardware or virtual machine (Veritas, 2019). The process of containerisation can encapsulate applications, their dependencies, and other resources into a single container. This process is not limited to applications, entire operating systems can be containerised.

Compared to traditional virtual machines, containers offer marked performance increase and minimisation of resource usage. One study concluded that container performance is comparable to that of a physical machine, with less than 4% overhead in terms of latency and throughput (Li et al, 2017). Containers also have features that make them more secure in comparison to virtual machines, such as shorter lifetimes. Compared to resource intensive virtual machines, this leads to containers being considered disposable. With lifetimes measured in hours, or minutes for specific applications, containers can be created as needed for as long as they are needed. A key security features is application isolation, which limits the possibilities of malicious activity penetrating an entire system (Veritas, 2019). Individual containers can have their own security levels set, depending on the requirements of that specific container.

Containers are also much more suitable to the development process as opposed to virtual machines. They are an excellent fit with current DevOps processes, and studies show that their development and management overall involves less time and effort on the part of administrators and developers (Maheshwari et al, 2018). This integration into the development process could lead to non-compliant containers quickly being modified and redeployed when necessary. A 2016 report also investigated the actual cost savings of using containers over virtual machines, and concluded that their reduced resource usage could actually lead to value creation for a business (Rogers and Lyman, 2016).

2.3.1. Container vulnerabilities

Despite all their advantages, containers do have their own security issues and vulnerabilities. These can extend from how the containers function, to the environments in which they are developed. Sysdig's 2019 container usage whitepaper showed Docker as the most used container platform being Docker, with 79% of surveyed organisations reporting its usage (Sysdig, 2019). As Docker is currently the most commonly used platform, container vulnerabilities have been explored through its container implementation and development ecosystem.

Docker containers themselves have been found to be secure, with this security greatly increased if they are run with non-privileged access and the underlying kernel it has accessed to hardened (Bui, 2015). Issues arise when the code used to develop containers is passed through external parties as part of the development process, greatly increasing the attack surface. However this risk and other issues could be reduced by the use of orchestration platforms to improve isolation, and possibly more abstraction to remove the majority of host dependence (Martin et al, 2018).

It appears that despite being largely secure as a technology, vulnerabilities can be exposed through the container development process. This necessitates an examination of how Docker containers are developed, and how an error or oversight leading to a vulnerability could slip though version control. An empirical analysis of Docker containers on GitHub yielded the following results relevant to this project (Cito et al, 2016):

- Most containers inherit infrastructure from large operating systems, negating any attempts to reduce a container's footprint.
- 28.6% of all quality issues were a result of version labelling being missing or incorrect.
- Dockerfiles dealing with dependencies are rarely updated.

These findings help explain how vulnerabilities can arise in containers. Inheriting infrastructure by essentially containerising an entire virtual machine not only reduces the container's overall performance, but also creates a larger attack surface simply by having

unnecessary components still present. It would be more prudent to containerise the application or service that was running on that virtual machine, reducing its attack surface and increasing performance. Errors in version labelling could result in containers with known vulnerabilities still being in use, with the misconception that any code used to build the container is the latest and most secure. Proper labelling would greatly reduce the occurrence of this, with the latest patches and updates being made readily available. The files containing dependencies could easily become outdated, resulting in what is otherwise a secure container relying on an unsecure component. In implementing and maintaining compliance standards, avoiding these mistakes could help address potential vulnerabilities and lead to a more secure container build. Regular testing and updates, properly labelled and made available with up to date dependencies, could assist in keeping a container compliant.

2.4. Virtualised infrastructure

In addition to operating systems and applications, network infrastructure can also be virtualised. Virtual switches have long been in use for virtual machines, for example, and dedicated virtual machines can provide necessary network services. Containerisation provides the opportunity to containerise these services, and deploy network services entirely as code. However, despite this being possible there can be significant downsides and trade-offs in security. For example, containers run on a single host are faced with a choice between better performance or better security. If making security and isolation of containers a priority, bridge mode will account for this requirement but decrease network performance. If network performance is the main priority, host mode will allow performance comparable to native performance (Suo et al, 2018). This balance between performance and security could be an issue when containerising network infrastructure in a compliant manner.

2.4.1. Software-defined networking

Software-defined networking (SDN) is a networking approach that handles network configuration programmatically. Traditional networking has static architecture, while SDN allows much more flexibility in development. This approach has been found to be more

efficient in both development and deployment, and allowing for innovation in infrastructure development (Kreutz et al, 2014). This makes it an ideal candidate for containerisation, as development could be further enhanced by the benefits containers can bring. One way to containerise networks is using Docker, which in addition to the above benefits also adds a degree of scalability to virtualised network infrastructure. Using Docker, an isolated SDN controller placed between the physical and virtual network allows network functionality to be customised as necessary, all while providing quick deployment and at lower cost than traditional methods (Xingtao et al, 2018).

An additional benefit of SDN is its multi-tenant nature, allowing each tenant to appear as if it alone exists on the network. This degree of isolation is a highly desirable feature when considering the secure transit and storage of data. The degree of flexibility provided by SDN allows each tenant to have their topology customised to meet specific requirements, and incorporates elements from containerisation in order to provide network scalability (Drutskoy et al, 2013). This means that the development of services and infrastructure can occur simultaneously, and with regular testing, can allow for the rapid deployment of upgrades to bring this containerised infrastructure into compliance.

2.4.2. Virtual network emulation software

In researching SDN and virtualised infrastructure in general, a useful tool called Mininet was discovered. It is a free network emulator, available at www.mininet.org. As a learning tool it proved invaluable, as one of its features is being able to create an entire virtualised infrastructure in seconds. Monitoring the traffic of a sample network configuration with Wireshark allowed for a great deal of experimentation to better prepare for a future implementation of SDN.

2.5. Automation of IaC

Automation allows what were once manual tasks to be scripted and deployed in significantly less time than the manual process. Virtualising infrastructure and services as containers makes them prime candidates for automation, though this is not without its challenges. As with software development in general, scripts can have defects and vulnerabilities. Attempts have been made to categorise and address these vulnerabilities, to better aid the code review process.

One such attempt resulted in the creation of a tool, ACID, to detect defects from a taxonomy of eight categories, finding configuration related defects to be most common among all IaC scripts processed by the tool (Rahman et al, 2019). The area least found to be defective, the idempotency of the IaC scripts, shows that features such as desired state configuration (DSC) tend to be effective when put in place. This is relevant as any container with an idempotent configuration that drifts into non-compliance can simply have this drift corrected and compliant settings restored. Another report confirmed that technical issues surrounding automated delivery of IaC scripts were more or less consolidated, with the majority of issues relating to code likely to arise due to a lack of effective collaboration (Leite et al, 2020).

The tools available to facilitate IaC largely fall into two categories, each with their own intended functions. Configuration orchestration tools, such as Terraform, are intended to automate infrastructure deployment. Configuration management tools on the other hand, such as Chef, are intended to deploy services on provisioned infrastructure. Both categories of tools typically have integration with popular platforms. Terraform for example can use the Docker API to deploy containers via automated scripts. Despite documentation and resources available to implement containers in this manner, studies show that a lack of standardisation in container technologies and monitoring prevents smaller businesses from adopting them for use (Tosatto et al, 2015). This could present an issue when considering containerisation to meet PCI-DSS compliance. The Open Container Initiative was an attempt to create industry standards for container formats and run times, which could eventually help resolve this issue.

2.6. Chapter Summary

This chapter covered an extensive literature survey into the areas within the scope of this project. Articles and journals relating to PCI-DSS, containers, virtualised infrastructure, and automation were reviewed. Particular attention was given to any features which may be advantageous in reaching project goals, as well as any findings that reveal potential issues. Overall it appears that the PCI-DSS standard is effective when implemented correctly, and containers can be implemented securely. Despite this, the literature survey shows this is not a certainty. Containerisation is still relatively new, and containerising whole infrastructures newer still. While not exhaustive, the reported findings should be enough to form a reasonable implementation of the research question, with appropriate technologies employed to assess the viability of an automated containerised infrastructure in meeting PCI-DSS requirements.

3. Implementation & Design

In order to answer the research question, a practical implementation based on the literature review was required. Based on the specific areas of technology researched, various tools were selected to develop this, with the end goal of providing a containerised infrastructure that could be assessed against PCI-DSS requirements.

3.1. Design considerations

In designing the practical element of this project, various factors based on availability, hardware capability, and reproducibility were considered. The specific tools chosen from the areas of interest researched in the literature view were selected based on the level of integration they have with each other, their availability, and their cost. The majority of software used is free and open-source, with the exception being Windows 10. However this operating system is widely available, and was not considered a detriment to the goal of creating a reproducible test environment.

In selecting specific tools, consideration had to be given to the physical limitations of available hardware. Where possible, software with low hardware requirements was utilised to prevent performance issues. In attempting to reproduce the test environment, providing the hardware used has at least a 1st generation Intel Core i5 processor and 8GB DDR3 RAM, there should be no issues in doing so.

Another factor considered was how suitable the software and tools selected for use are in themselves being PCI-DSS compliant. As a requirement to being compliant with PCI-DSS is building secure systems, it was important to research the security and known issues with these choices using industry and academic sources. While the focus of this project is assessing the containerised infrastructure itself, the technology it is built on should not be ignored.

3.2. System context diagram

The following image is a simplified graphical depiction of the test environment created for use in this project. It depicts the Windows 10 host machine, encapsulating Hyper-V and a VM. Within this VM is the Terraform install, which then deploys Docker and its containers. One of these containers is MySQL Server, one an Alpine Linux-based container acting as firewall for the host virtual machine, and the third container is phpMyAdmin, a web-based frontend for MySQL Server. Once deployed, the web frontend can be access from the web browser on the Docker host.

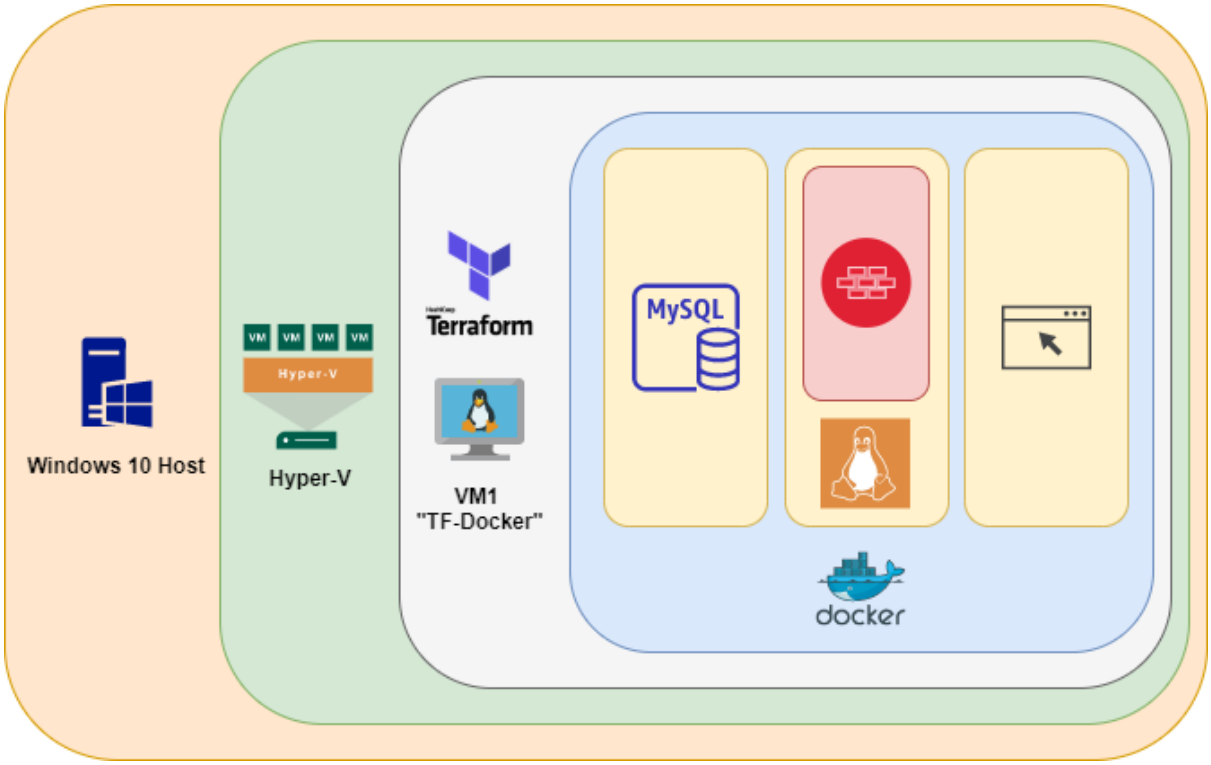

Figure 1: System Context Diagram

3.3. Software selections

Based on the conducted literature review, the following software was selected as the basis for the test environment of this project. Their selection will be justified based on what was learned in the literature review, and from research into the individual software itself:

- Docker.
- Terraform.
- Lubuntu 20.04.
- Windows 10 Hyper-V.

3.3.1. Software selection – Docker

Docker is an open-source platform used for the creation and management of containers. The Docker Engine consists of three components allowing the interaction of containers with other containers and networked resources: the 'docker daemon', REST API, and CLI client (Docker, 2020a). The CLI client uses the REST API as a bridge to the 'docker daemon' to initiate and manage containers. This 'daemon' can be considered similar in function to a server, where the various Docker objects are created. Docker was chosen as the containerisation platform for this project due to its availability, robust documentation and support, as well as it integration with other tools chosen for use in this project. In addition, Docker holds a sizeable market share when compared to other containerisation platforms at 23% as of 2019 (Galabov, 2019), making it a valuable tool to become familiar with.

In terms of security, Docker has been found to be a secure platform when configured correctly. It relies on features of the Linux kernel to isolate its containers, making using of 'namespaces' and 'cgroups' (Combe et al, 2016). By separating processes and namespace, Docker allows containers to view each other as remote hosts, further increasing the degree of isolation from each other (Catuogno and Galdi, 2016). The only notable issue with Docker's default isolation configuration is that containers use the same network bridge, named 'docker0'. This configuration can be changed to further isolate containers.

This level of isolation is desirable when creating an infrastructure that is PCI-DSS compliant. Having multiple services coexist on the same device without direct interaction unless specified, means that access and traffic can be tightly controlled. It can also help meet requirements for physical security and reduce expenditure, by reducing the overall amount of hardware needed to support and secure essential services.

3.3.2. Software selection – Terraform

Terraform is a change configuration and orchestration platform for deploying and versioning infrastructure as code, using configuration files to generate execution plans for the desired infrastructure (HashiCorp, 2020a). These execution plans and the language used in them are declarative, meaning they specify the intended goal but not the procedural steps to reach this goal (HashiCorp, 2020b). The language itself contains 'blocks' which contain the 'arguments' that assign values a name, and 'expressions' that either reference, combine, or literally define a value. These 'blocks' are saved as '.tf' files, ready to be used by Terraform.

This platform was chosen as it has plugins called 'providers', which Terraform uses to interact with APIs and resources. The Docker provider allows Terraform to create containers and manage images using the Docker API (HashiCorp, 2020c). Using this provider, it is possible to deploy Docker containers to any number of machines. The advantage of doing so with Terraform instead of Docker by itself is automation. A configuration script would allow the deployment of a containerised infrastructure, on any number of machines required. This allows for proper version control, with Terraform taking care of the actual implementation of the desired infrastructure. It also has the advantage of being free and open-source software, available on a variety of operating systems.

Compliance for Terraform would likely involve following best practice as described in official documentation. Described as 'Compliance as Code', this approach takes standard practices such as proper version control and preventing manual changes to infrastructure, and incorporates them into official best practice (HashiCorp, 2020d). As the platform and its competitors are relatively new, suitable documentation and reference material can be difficult to come across.

3.3.3. Software selection – Windows 10 Hyper-V

While not strictly necessary to meet the goals of this project, the use of a virtual machine made reproducing the test environment a simple matter. Hyper-V itself is available as a component in the Professional Edition of Windows 10, and while not free software it is widely available. This software was chosen due to its availability, and because other virtualisation programs were incompatible with available hardware. In order to fulfil PCI-DSS requirements the underlying system running Hyper-V was hardened using best practice and other secure configuration changes (Zamora et al, 2019).

3.3.4. Software selection – Lubuntu 20.04

Lubuntu is a spin of Ubuntu, designed for low-end computers. This operating system was chosen to be used in conjunction with Hyper-V to deploy and host a containerised infrastructure. While Docker and Terraform are cross-platform, both originate from Linux, and the majority of available documentation and support is primarily aimed at Linux usage. Best security practice for Ubuntu-based systems was applied to all virtual machines (Canonical, 2020). As this component of the project is the host for the containerised infrastructure, hardening it according to best practice was considered necessary to meet PCI-DSS requirements.

3.4. Test environment

The test environment consists of one Lubuntu virtual machine, and the Windows 10 host it is hosted on. From the Windows 10 host, the VM is created and Terraform installed to deploy the Docker network within it. The virtual machine is placed on an external network in Hyper-V, allowing internet connectivity. The commands needed to recreate portions of this environment have been included in document where appropriate.

3.4.1. Initial Windows 10 host configuration

While Windows 10 Pro is capable of running Hyper-V, this is not enabled by default. The Hyper-V platform and its management tools must be enabled in Control Panel or by using PowerShell. Once installed and the host restarted, Hyper-V Manager is available and PowerShell is capable of successfully running VM-related commands. The latest Windows Update patches were applied, and system information recorded. Using the command prompt, system information was logged using the 'systeminfo' command:

```
Host Name:                 DESKTOP-6C47R1D
OS Name:                   Microsoft Windows 10 Pro
OS Version:                10.0.18363 N/A Build 18363
OS Manufacturer:           Microsoft Corporation
OS Configuration:          Standalone Workstation
OS Build Type:             Multiprocessor Free
Registered Owner:          [redacted]
Registered Organization:
Product ID:                00330-80000-00000-AA980
Original Install Date:     10/04/2020, 23:44:06
System Boot Time:          02/07/2020, 18:18:51
System Manufacturer:       Hewlett-Packard
System Model:              HP Pro 3130 Microtower PC
System Type:               x64-based PC
Processor(s):              1 Processor(s) Installed.
                           [01]: Intel64 Family 6 Model 30 Stepping 5 GenuineIntel ~2668 Mhz
BIOS Version:              American Megatrends Inc. 6.14, 05/11/2010
Windows Directory:         C:\Windows
System Directory:          C:\Windows\system32
Boot Device:               \Device\HarddiskVolume1
System Locale:             en-us;English (United States)
Input Locale:              en-ie;English (Ireland)
Time Zone:                 (UTC+00:00) Dublin, Edinburgh, Lisbon, London
Total Physical Memory:     8,151 MB
```

Figure 2: System Information CMD partial output

The full output was saved as a text file with personal information redacted, which can be found in the appendices. At this point a system restore point was created and saved externally, for use in the event the host machine required its operating system and settings reinstalled. The following PowerShell command will enable Hyper-V:

Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -All

Link to full 'systeminfo' output:
https://github.com/danecode/lyit-terraform-project/blob/master/systeminfo_full.txt

3.4.2. Hyper-V setup and configuration

By default, Hyper-V provides a virtual switch which provides access to the host and its network via Network Address Translation (NAT) (Microsoft, 2016a). As well as internet access being required to install packages on the Linux VM, this level of connectivity was required to test how isolated the deployed Docker containers are from other networked resources.

Setup of the virtual machine required the download of the latest 64-bit Lubuntu 20.04 image, the link to which is available in the appendices. Once obtained a standard Generation 2 VM was created, with dynamic memory enabled and a new 10GB dynamically expanding virtual disk attached. Generation 2 VMs provide support for UEFI and modern security features (Microsoft, 2016b).

A requirement of Secure Boot in Hyper-V for running Linux VMs is the use of a different security template, otherwise Linux VMs will not boot (Microsoft, 2014). Setting the VM to use the 'Microsoft UEFI Certificate Authority' will allow the VM to boot as normal. Enabling Secure Boot prevents unauthorised code from running at boot. The available encryption options were not enabled, as the Lubuntu VM was to be hosted solely on the Windows 10 host and not migrated to any other devices.

3.4.3. Virtual machine configuration

Starting the Lubuntu VM with the Lubuntu ISO attached will cause it to boot as if using a live CD on physical hardware. The operating system will boot into its live environment, and from there can be installed. Since internet connectivity is enabled by Hyper-V using NAT, the operation system installer will detect language and regional settings. From here the installer will detect the virtual hard disk, which will be formatted and used by Lubuntu. Once the user

details, root password, and computer name are set, the installation is complete and the VM will restart. On restart, providing internet connectivity is still active, Lubuntu will attempt a full system upgrade. If this notification does not appear or otherwise, the following command will update the package indexes and initiate the upgrade:

```
sudo apt-get update && sudo apt-get upgrade
```

At this point it is recommended to take a checkpoint of the Hyper-V VM, in the event configuration errors necessitate resetting the test environment. It is also useful for repeated testing, allowing for experimentation with different approaches and technologies if desired. The final stage of VM configuration is installing the Terraform and Docker packages, which can be done using the following commands:

```
sudo snap install terraform
sudo apt install docker.io
```

To enable Terraform to access the Docker daemon, create a file name ‘daemon.json’ in the ‘/etc/docker’ folder with the following contents:

```
{"hosts": ["tcp://0.0.0.0:2375", "unix:///var/run/docker.sock"]}
```

Finally, start the Docker daemon with the following command:

```
sudo dockerd
```

After creating another Hyper-V checkpoint, the VM configuration can be considered complete and ready to deploy the containerised infrastructure.

3.4.4. Containerised infrastructure overview

The containerised infrastructure will run a number of typical network services and applications that can be assessed using PCI-DSS compliance requirements. As PCI-DSS is primarily concerned with the secure storage of credit card information, the main service

running on the structure will be a database container and its attached storage. The specific Docker images needed to achieve this have been selected based on what was learned from literature review, and from research into the images and their capabilities. The Docker images and functions used are:

- MySQL Server – a containerised version of the database software MySQL Server.
- A persistent storage volume created and managed by Docker to store data.
- phpMyAdmin – a containerised version of phpMyAdmin, which allows browser-based administration and use of MySQL databases.
- iptables – a containerised version of the 'iptables' firewall application based off an Alpine Linux container image.

3.4.5. Docker storage methods

'Volumes' are a method Docker uses to store persistent data, that are completely managed by Docker (Docker, 2020b). These volumes can be attached to any container, and be accessed by multiple containers simultaneously as shared storage. Another method to store persistent data is by using 'bind mounts', which are directories accessible on the Docker host mounted by a container (Docker, 2020c). However using 'bind mounts' would not meet the aims of this project, which to assess a containerised infrastructure against PCI-DSS compliance standards. Having the host involved in anything outside of hosting Docker and its containers would be outside the scope of this project. The Docker volume is used solely to store MySQL databases. With this consideration and the fact volumes exist as part of Docker's private storage space, this can be considered a reasonable choice in setting up secure storage to meet PCI-DSS requirements. The Docker volume 'mysql_db' was created in the same block of commands that create the MySQL container, and is attached to this container at its creation.

3.4.6. Docker network configuration

By default, Docker uses 'bridge' networks for containers created by it, which enables network segments to forward traffic between each other (Docker, 2020d). These 'bridge' networks allow containers on the same host to communicate, providing they are joined to that specific network. The advantage to using a user-defined bridge network is that name

resolution is configured by default, allowing containers to communicate with each other with no issues. A now deprecated method in linking containers for connectivity is the '--link' flag which would allow full communication but also leave that container's ports exposed (Docker, 2020e). When deploying the container images used in this project, both were assigned to a user-defined network to avoid using this deprecated and insecure method. Instead, a custom bridge network named 'dbnet' was created with a subnet of 172.20.0.0/16, and the only ports published for the containers were ones necessary for them to function. 'MySQL Server' functions by default over TCP port 3306, and 'phpMyAdmin' over TCP port 80. In the case of 'phpMyAdmin', this was mapped to TCP port 8080 on the host machine. No other changes to the network were made, and as is examined in a later section of this chapter, setting up Docker security within containers proved problematic. The network adapter created by Docker was also named 'dbnet', instead of allowing Docker to assign it a random name. This was to ensure later commands would function when passed to the shell from the automated script.

IP addresses of created containers are assigned automatically by Docker, based on the configuration of the network they are attached to. However these were statically assigned in the Terraform script for consistency. As the data volume is persistent, the containers themselves can be destroyed and recreated as needed. In terms of security and meeting PCI-DSS requirements, this could be considered as a secure system and secure information management. The services running in these containers can be destroyed when not needed, and therefore unable to be compromised. The only concern in this event is the security of the persistent data volume.

3.4.7. Alpine Linux based containers

Alpine Linux is a lightweight, hardened Linux-based operating system with a focus on security. A container running Alpine Linux can be as small as 8MB, making it suitable for use in containers due to its efficient use of resources. With security as a primary design consideration, all applications available for Alpine Linux have 'stack smashing' protection, and are compiled as Position Independent Executables (PIE) (Alpine Linux, 2020). 'Stack smashing' is a form of attack used to exploit vulnerabilities in a system's buffer overflow, with the intention of seizing control from the application under attack (Chen et al, 2015).

There are various methods to defend against a 'stack smash', with one of these being the use of Position Independent Executable (PIE) binaries. PIE binaries are the result of a hardened build process, with a binary and its dependencies being run in random memory locations which differ on each new execution (Red Hat, Inc, 2012). While an effective measure against 'smash attacks', evolving attack methods have made position-independent code reuse attacks a reality, which can target programs based on their relative location in memory (Göktas et al, 2018). While the threat of attack is always a concern for any system, Alpine Linux's focus on security in all aspects of its design makes it a reasonable choice for a container base, and inclusion in a network aiming to meet compliance requirements.

3.4.8. Container image – iptables

'iptables' is a firewall package available for Linux distributions, allowing the creation and management of firewall rules on a system. This application was run in a container based on Alpine Linux, and configured to allow access to TCP ports 8080 and 3306 on the 172.20.0.0/16 network. This container was not joined to the 'dbnet' network, rather it was attached to the 'host' network to allow firewall rules to be set for the host machine, and by extension any service that runs on it. While breaking container isolation by accessing the host's network space, this does provided a containerised security solution for the entire system. An additional rule needed to be added to 'iptables', to allow the firewall to accept connections from the 'dbnet' network interface.

3.4.9. Container image – MySQL Server

MySQL Server is an open-source, cross-platform relational database management system. Using built in encryption functions, it is possible to encrypt stored values using a key (Oracle, 2020a). Doing so prevents access to encrypted data if an attacker somehow gains access to a client running MySQL and connected to the encrypted database. As every query with encrypted data remains so without entering the key, this data will still be protected. Regarding general security measures and best practice, official documentation is available detailing how to secure a MySQL Server installation and run it safely in a production environment (Oracle, 2020b). For this project, MySQL Server was run in its own container, with the actual database stored in a separate Docker volume. The server and database were intended to be accessed from a VM client running a MySQL client.

When creating the MySQL Server container, its default TCP port 3306 is published to enable communication and the administration and modification of its databases. A blank database named 'db' was created, as well as two users. One is the default 'root' account, which has complete control over MySQL Server and its contents. The other, 'user1', is a user with less privileges to modify the database. They have read-only access to 'db', and cannot modify or create other users. This variable level of access to sensitive information can be considered as secure user configuration, and appropriate access to resources in meeting PCI-DSS compliance requirements.

3.4.10. Container image – phpMyAdmin

'phpMyAdmin' is a web-based frontend for MySQL Server. This was chosen instead of a MySQL client as it does not require the installation of additional software on any hosts, and can be accessed through any modern web browser (phpMyAdmin, 2020). Using a dedicated client can increase the complexity of deployed infrastructure, with reliance on host and server being functional to provide the necessary service. By moving this service into a container running a web-based client, it ensures that only an internet or intranet connection is required for access. As the information it actually accesses is stored elsewhere in a persistent volume, this web client can be destroyed and recreated as many times as necessary.

In order to function, 'phpMyAdmin' requires access to the MySQL Server container. This can be done with '--link' flag as discussed previously, but would expose the MySQL Server container far more than necessary. By connecting the 'phpMyAdmin' and MySQL Server containers on the same user-defined bridge network, they can communicate as intended and without exposing any unnecessary ports. Once deployed, 'phpMyAdmin' can be accessed by the address 'http://172.20.0.3' in any web browser. From there, depending on which user account is used to login, the MySQL database and settings can be viewed or modified.

3.4.11. Firewall and security issues

Significant issues were encountered in attempts to deploy containers with network security functions. Initially, the deployed infrastructure was intended to use an Alpine Linux based container for routing and firewall services. However due to how bridged Docker networks function, a dedicated router was deemed unnecessary. The deployed containers could be accessed via their published ports from the host, but this did not remove the need for security measures. There appear to be known issues with how Docker manages its IP tables for firewall services, which have been documented as overwriting any firewall settings in a container (Docker, 2020f). Various workarounds were attempted with no success. Images exist on Docker Hub that have preconfigured Docker images consisting of 'iptables' and proxy services, such as the image used in this project. A solution to the issue was found by attaching the 'iptables' container to the host network space, allowing it to manage firewall rules for the entire host. These issues with creating containers for security services are important to note when considering the containerisation of network infrastructure. It may be better to leave some services running on dedicated devices, or in the case of virtualised services, secure the host they are running on.

3.5. Terraform script design

To design the Terraform script to deploy the containerised infrastructure, setup was first conducted manually on the Lubuntu virtual machine. Images were pulled manually, and various implementations of the container images were trialled. It was during this stage that the deprecated '--link' flag and issues regarding firewalls were discovered. Once the Docker

commands used were confirmed to be functional, and the containers working as intended, the variables and settings passed in these commands were used in the Terraform script. Firstly, a new bridged network was defined and necessary images pulled from Docker Hub:

```
sudo docker network create --opt com.docker.network.bridge.name=dbnet dbnet \
--driver=bridge --subnet=172.20.0.0/16 dbnet
sudo docker pull mysql/mysql-server:latest
sudo docker pull phpmyadmin/phpmyadmin:latest
sudo docker pull paulczar/iptables:master
```

A firewall container, 'firewall', was created from the 'iptables' image and connected to the host's network space:

```
sudo docker run --name firewall --net host -e TCP_PORTS=8080,3306 -e HOSTS=172.20.0.0/16 \
--cap-add=NET_ADMIN paulczar/iptables:master
```

An additional rule needed to be added to 'iptables' to allow the firewall to accept connections from the 'dbnet' interface, run inside the 'firewall' container:

```
sudo docker exec -it firewall /bin/ash/
iptables -A INPUT -i dbnet -j ACCEPT
```

Two containers, 'mysql_server' and 'php_admin' were then created and attached to the 'dbnet' bridged network:

```
sudo docker run --name mysql_server --network dbnet -e MYSQL_ROOT_HOST=% -e MYSQL_DATABASE=db \
-e MYSQL_ROOT_PASSWORD=root -e MYSQL_USER=user1 -e MYSQL_PASSWORD=user1pass -d -p 3306:3306 \
--mount  source=mysql-db,target=/var/lib/mysql mysql/mysql-server:latest
```

```
sudo docker run --name php_admin --network dbnet --network dbnet -e PMA_HOST=mysql_server -d \
-p 8080:80 phpmyadmin/phpmyadmin:latest
```

The specific details of these commands will be examined in the analysis of the Terraform script itself.

3.5.1. Terraform script analysis

The Terraform script, the design of which was informed by Docker commands, provides the same functionality as manual deployment but with minimal effort. By automating the entire process, the containerised infrastructure can be rebuilt as needed in much less time than manual configuration. A thorough breakdown of the completed Terraform script, 'ddeploy.tf', demonstrates how this is accomplished.

```
provider "docker" {
  host = "tcp://127.0.0.1:2375/"
}
```

Code Listing 1 Docker Provider and information on host

Code Listing 1 - 'Providers' are used by Terraform to interact with various services and plugins, in this case, Docker. Terraform pulls the provider from the internet based on its name, "docker", and searches for access to the Docker daemon on the host network over TCP port 2375 as defined in the 'daemon.json' file.

```
resource "docker_image" "iptables" {
  name = "paulczar/iptables:master"
}

resource "docker_image" "mysql-server" {
  name = "mysql/mysql-server:latest"
}

resource "docker_image" "phpmyadmin" {
  name = "phpmyadmin/phpmyadmin:latest"
}
```

Code Listing 2 Pulling Docker images from Docker Hub

Code Listing 2 – These three code blocks create Terraform resources from images on Docker Hub. "docker_image" lets Terraform know what the resource in the curly braces is. The values of the 'name' variable consist of the Docker Hub account and the specific image found by its tag.

```
resource "docker_network" "private_network" {
  name = "dbnet"
  driver = "bridge"
  options {
    "com.docker.network.bridge.name" = "dbnet"
  }
  ipam_config {
    subnet = "172.20.0.0/16"
  }
}
```

Code Listing 3 'dbnet' bridged network configuration

Code Listing 3 – a Docker bridge network is created named 'dbnet', with the network interface given a definitive name of 'dbnet' rather than one randomly assigned by Docker. The 'dbnet' network has been given a subnet of 172.20.0.0/16, and any container joined to the network will be assigned an address from this IP range.

```
resource "docker_container" "iptables" {
  name = "firewall"
  image = "${docker_image.iptables.latest}"
  restart = "always"
  network_mode = "host"
  env = [
    "TCP_PORTS=8080,3306",
    "HOSTS=172.20.0.0/16"
  ]
  capabilities {
    add = ["NET_ADMIN"]
  }
  command = ["/bin/ash", "iptables -A INPUT -i dbnet -j ACCEPT"]
}
```

Code Listing 4 'firewall' container created from Alpine Linux-based container

Code Listing 4 – the ‘iptables’ image is used to create a container named ‘firewall’. The container is attached to the host network with the special ‘network_mode’ variable, and set to always restart. Its allowed ports are defined, and additional commands are passed directly to the containers shell to allow traffic to the ‘dbnet’ network. The ‘capabilities’ block and its ‘NET_ADMIN’ value allow these rules to take effect on the host.

```
resource "docker_container" "mysql-server" {
 name = "mysql_server"
 image = "${docker_image.mysql-server.latest}"
 restart = "always"
 networks_advanced {
  name = "dbnet"
  ipv4_address = "172.20.0.2"
 }
 env = [
  "MYSQL_ROOT_HOST=%",
  "MYSQL_DATABASE=db",
  "MYSQL_ROOT_PASSWORD=root",
  "MYSQL_USER=user1",
  "MYSQL_PASSWORD=user1pass"
 ]
 mounts {
  source = "mysql_db"
  target = "/var/lib/mysql"
  type = "volume"
 }
 ports {
  internal = 3306
  external = 3306
 }
}
```

Code Listing 5 ‘myql_server’ database server configuration

Code Listing 5 – The database container ‘mysql_server’ is created from the ‘mysql-server’ image. A blank database is created, along with the root password and a standard user account, ‘user1’. These values can be changed later through the web interface. A persistent

volume is created named ‘mysql_db’ and attached to the container. This volume will persist even if the container is destroyed and rebuilt. TCP port 3306 is published, to allow MySQL traffic.

```
resource "docker_container" "phpmyadmin" {
 name = "php_admin"
 image = "${docker_image.phpmyadmin.latest}"
 restart = "always"
 networks_advanced {
  name = "dbnet"
  ipv4_address = "172.20.0.3"
 }
 env = [
  "PMA_HOST=mysql_server"
 ]
 ports {
    internal = 8080
    external = 80
 }
}
```

Code Listing 6 ‘php_admin’ web-based frontend for MySQL Server

Code Listing 6 – A ‘phpmyadmin’ image is used to create the ‘php_admin’ container, which runs a web-based client to access MySQL Server databases. From this interface, databases and accounts can be managed and modified. The environment variable ‘PMA_HOST=mysql_server’ is required to point the web client towards the database server. It can be accessed from any browser using the address ‘http://172.20.0.3’.

3.5.2. Summarised setup and running the Terraform script

To summarise the steps to prepare the test environment, see the following list of commands. All files mentioned are uploaded to Github, and links attached where appropriate, and also included in the appendices of this document.

1. From the terminal run "*sudo apt-get update && sudo apt-get upgrade*", "*sudo snap install terraform*", "*sudo apt install docker.io*" in that order.
2. Either download '*daemon.json*' or create in '*/etc/docker*' with root privileges with the following contents: *{"hosts": ["tcp://0.0.0.0:2375", "unix:///var/run/docker.sock"]}*
3. Back in the terminal, run "*sudo dockerd*" to start the Docker daemon.
4. Use the '*cd*' command to change directory to the location you have saved '*ddeploy.tf*'.
5. Run the following commands and confirm, and Terraform will deploy the containerised infrastructure: "*terraform init*", "*terraform plan*", "*terraform apply*".

Providing the above steps were followed in the designed test environment, barring unforeseen errors Terraform will deploy the containerised infrastructure as designed.

Link to 'daemon.json':
https://github.com/danecode/lyit-terraform-project/blob/master/systeminfo_full.txt

Link to 'ddeploy.tf': https://github.com/danecode/lyit-terraform-project/blob/master/ddeploy.tf

3.6. Evidence of deployment and additional configuration

All output from the console commands was logged as a text file, and uploaded to Github. The links to specific files are included after relevant screenshots, and in the appendices of this document as a list. The screenshots referenced by the name 'Figure/s' are located in the Appendix C of this document.

Figure 3 shows the entire output of the 'terraform init' command, which was run from terminal on the VM host Desktop. This was where 'ddeploy.tf' was located for the testing phase of this project. Terraform downloads the Docker provider, and initiates itself for use.

Figures 4-6 depict the partial output of the 'terraform plan' command. At this stage, Terraform is analysing the 'ddeploy.tf' file to determine the changes to be made to infrastructure. If errors are detected in the script, a warning will appear noting the nature of the error. Resources that are scripted correctly will appear green in the console, with their information and variables displayed. Figure 6 depicts the total number of resources that will be deployed based on the plan. The complete terminal output can be accessed here: https://github.com/danecode/lyit-terraform-project/blob/master/terraform_plan.txt

Figures 7-8 depict the partial output of the 'terraform apply' command. Based on the plan developed from 'ddeploy.tf', Terraform will now attempt to deploy the infrastructure. In the case certain resources have been scripted incorrectly, those that are scripted correctly will still be deployed. Terraform informs the user that correcting the script and running 'terraform apply' again will deploy only what it detects to have been changed in the script. This allows for an iterative approach to debugging and updating the script to run as intended. The complete output of the 'terraform apply' command was saved to a text file, which was uploaded to Github. It can be accessed at the following link: https://github.com/danecode/lyit-terraform-project/blob/master/terraform_apply.txt

Figure 9 depicts the results of a 'docker ps' command with custom format to display vertically, which shows the containers deployed from the Terraform script. Figure 10 depicts an interactive terminal run in the 'firewall' container, which shows the firewall rules for the 'iptables' package using the 'iptables -S' command.

Figure 11 depicts the 'phpMyAdmin' frontend login page, accessed in Mozilla Firefox using the address 'http://172.20.0.3/'. Once logged in using the 'root' account, the user is taken to the main page of the frontend, where options and databases can be accessed, as depicted by Figure 12.

Figure 13 depicts changing the permissions for the account 'user1', restricting them to SELECT statements and creating and using views. This is to prevent an ordinary user having an unnecessary level of privilege when accessing sensitive information. Figure 14 depicts changing the 'root' account password from its default defined in 'ddeploy.tf'.

Figure 15 depicts a user on the Lubuntu VM attempting to access the Docker volume through the file manager, and being denied accessed. This can be considered secure storage, however Figure 16 depicts the contents of the Docker volume being accessed by a user with 'root' privileges.

3.7. Chapter Summary

This chapter detailed the specific technologies and design choices of the practical implementation of this project. Based on the findings of the literature review, suitable technologies were selected for use in constructing a test environment to host a containerised infrastructure. With the test environment in place, it is possible to assess the resulting configuration against PCI-DSS compliance requirements. Diagrams, screenshots, and output logs were provided to demonstrate the setup of the infrastructure, as well as evidence of its functionality.

4. Results and Testing

With the containerised infrastructure successfully deployed via automation, it becomes possible to assess the suitability of this execution in meeting PCI-DSS compliance requirements. As the focus of this project is on the twelve high-level requirements of PCI-DSS, these have been used as the basis for a qualitative approach to testing. The PCI-DSS documentation provides testing guidelines that will be utilised where applicable (PCI-SSC, 2018b). The guidelines will be applied to the design as it was implemented, with any requirements failures being addressed by design improvements, and by taking into account the test environment.

4.1.1. Req. 1 – Install and maintain a firewall configuration to protect cardholder data

A firewall was successfully installed and configured, being deployed by Terraform as a Docker container. This firewall shared the host's network namespace, and so its firewall rules were applied system-wide and not just to other containers. While this does function and meet the requirement for a firewall configuration, having the firewall work exclusively within the Docker network proved problematic. The solution to this was to give the firewall container system-wide access to prevent iptable conflict from Docker overwriting any settings specified by the automated script.

The firewall container as implemented is functionally no different from configuring a firewall directly on the Docker host. Considering the problems implementing a container-based firewall to protect cardholder data, directly configuring a firewall on Docker host may be a simpler way to meet this requirement. However this would mean that a compliant infrastructure would only be partly containerised.

4.1.2. Req. 2 – Do not use vendor-supplied defaults for system passwords and other security parameters

The 'ddeploy.tf' script contains placeholder passwords for both 'root' and 'user1' accounts for the MySQL Server database. This is necessary to provide a simple password that can be immediately changed once the infrastructure is deployed. Without specifying a value, a random password will be generated. This generated password is stored in a log file, which exists on the machine hosting the Terraform platform. Depending on who has access to this machine, and where the infrastructure is actually being deployed, this could be a security concern. It was decided to manually assign a password in the script, to take advantage of Terraform's ability to quickly destroy and rebuild deployed resources when required. Applying a temporary password of some sort is required for initial setup of resources, and as long as this is changed before the deployed systems are put into a production environment this can be considered as successful in meeting this requirement.

4.1.3. Req. 3 – Protect stored cardholder data

Cardholder data is stored within a Docker volume, accessible through a web client. This client is protected by configured 'root' and 'user1' accounts with appropriate levels of access for business use within the MySQL Server. However the Docker volume itself is stored on the Docker host, and a privileged user could access its contents. This is a vulnerability of the test environment, as the infrastructure and web client are being run on the same machine. Even if the web client was accessed from another machine, the Docker volume could still be accessed on the Docker host. Depending on physical and network security, this could be a serious security concern. The risk can be mitigated somewhat by encrypting sensitive data within the database, but this still leaves some data exposed. The exposure of any personal or sensitive information.

These risks can mitigated somewhat by using the security features of Hyper-V to encrypt and shield VM information from the Hyper-V host, however within the Docker host VM these vulnerabilities still exist. The VM cannot be completely locked down, as this would prevent all access to the web client and database server. With the current implementation, this cannot be considered as successful in meeting this requirement.

4.1.4. Req. 4 – Encrypt transmission of cardholder data across open, public networks

As the test environment involved only one machine in a non-production environment, this requirement cannot be fully tested. The current implementation was designed with the intent of being used on an internal system, without access to public networks. However the VM running the Docker host can have its traffic encrypted when it is being migrated to another server. The current implementation's networking and configuration requires reviewing and possible testing over a public network. Compliance with this PCI-DSS requirement is inconclusive, and requires additional investigation.

4.1.5. Req. 5 – Protect all systems against malware and regularly update anti-virus software or programs

The operating systems and components of the VM host and Docker host were updated with the latest security patches available. The actual images used by Terraform to build Docker containers were pulled from Docker Hub, the official source for such images. The use of Alpine Linux-based container images provided an acceptable level of security for these containers, Alpine being a hardened and lightweight Linux-based operating system design around security. It is a popular and well-documented base for Docker containers, and proved effective in this project. The firewall container provides security for the Docker host and containers, and as the host is also a Linux operating system, a dedicated anti-virus program is not required. It may be in future that malware and viruses are designed to target Linux more than Windows, but for the present day this is a reasonable decision. As for the Hyper-V host, which is running on Windows, appropriate anti-virus and anti-malware applications were installed and updated with the latest virus/malware definitions. Considering these factors, the current implementation can be considered successful in meeting this PCI-DSS compliance requirement.

4.1.6. Req. 6 – Develop and maintain secure systems and applications

Vendor-supplied security patches for all components were installed where possible. Consistent monitoring and updating of these patches as they are released will help to maintain secure systems and applications. The latest vulnerabilities and security concerns of the implemented technologies can be taken into consideration when gathered from

reputable sources, such as official websites or CVE analyses. Part of meeting this compliance requirement is the continual maintenance of developments in both the deployed services and new security threats. As long as this is process is ongoing, this implementation can be considered as successful in meeting this PCI-DSS compliance requirement. However, considering new security threats and vulnerabilities occur frequently, this compliance is subject to change.

4.1.7. Req. 7 – Restrict access to cardholder data by business need to know

Direct access to the MySQL database requires an authenticated user account. The 'root' account is capable of modifying all aspects of the database and configuration, while the 'user1' account has been limited in its capabilities. In this way access to sensitive information is restricted based on what the account is required to be used for. Direct access to the MySQL Server through a browser is not possible, authenticated credentials are still required. This current implementation can be considered successful at meeting this PCI-DSS compliance requirement. However the system can only remain compliant as long as user accounts are managed correctly, with an emphasis on restricting access exactly to only what is required for that account.

4.1.8. Req. 8 – Identify and authenticate access to system components

Unique user accounts are used to log in to system components. In this way modifications and usage of MySQL Server and contents can be monitored. In deploying this system to a production environment, these accounts would most likely be tied to a single sign-on account, further increasing the level of monitoring and authentication. As the test environment stands, this requirement cannot be fully tested without modification. For what has actually been deployed in the test environment, access to the MySQL Server via the phpMyAdmin frontend can be considered acceptable in meeting this PCI-DSS compliance requirement.

4.1.9. Req. 9 – Restrict physical access to cardholder data

Considering this is a college project, this requirement is largely untestable due to it not being conducted in premises with secure locations accessed by ID card or passcode. There were also no printers or other methods present to remove data from the test environment.

4.1.10. Req. 10 – Track and monitor all access to network resources and cardholder data

The 'Binary Log' accessible in the phpMyAdmin web client interface contains a log of all queries, errors, and other changes made to databases by logged in users. Using this log, and the other logs available, it is possible track and monitor all changes and access to sensitive information. Considering this capability, the deployed infrastructure can be considered successful in meeting this PCI-DSS compliance requirement.

4.1.11. Req. 11 – Regularly test security systems and processes

As vulnerabilities are discovered and updates are released, the Terraform script can be updated accordingly to reflect changes since the initial deployment. Deployed resources that do not require changes will be unaffected by Terraform updating the resources, only altering those resources that require changes. This feature of the Terraform platform also allows for quick and extensive testing to be carried out. Changes in infrastructure can be trialled using a branch of the deployment script, either in parallel to the production environment or in a separate test environment. As long as proper version control is used to preserve the integrity of the deployment script for the production environment, this implementation can be considered successful in meeting this PCI-DSS requirement.

4.1.12. Req. 12 – Maintain a policy that addresses information security for all personnel

As this project was conducted by one person in a test environment, the feasibility of implementing an information security policy to the extent required by PCI-DSS would not be practical. If version control using Github can be considering security for code revisions, then the deployed infrastructure could partially meet this requirement. However, with the current implementation this PCI-DSS requirement is largely untestable without an extensive expansion of resources. The use of Terraform as a platform and containers for services could be considered as partial fulfilment for an incidence response plan, as they are able to rebuild and redeploy infrastructure that has been tampered with or hacked, or requires immediate patching in a matter of minutes.

4.2. Results categorisation

Assessing the containerised infrastructure against PCI-DSS compliance requirements yielded interesting results. While meeting the majority of the high-level requirements, there was considerable ambiguity concerning this implementation. Part of this was due the nature of the test environment, and resource availability. Requirements generally fall into three categories based on the results of this implementation; 'successful', 'unsuccessful', and 'untestable under current conditions'. These three categories and which requirements fall under them will be discussed, as well as what this says overall about the suitability of the designed implementation in answering the research question.

4.2.1. Successful implementation of requirements

Of the twelve high-level PCI-DSS requirements used as assessment criteria, the implementation of a containerised infrastructure was deemed to have met seven of them:

- 1. Install and maintain a firewall configuration to protect cardholder data.
- 2. Do not user vendor-supplied defaults for system passwords and other security parameters.
- 5. Protect all systems against malware and regularly update anti-virus software or programs.
- 6. Develop and maintain secure systems and applications.
- 7. Restrict access to cardholder data by business need to know.
- 10. Track and monitor all access to network resources and cardholder data.
- 11. Regularly test security systems and processes.

Interesting to note is that all the above requirements were fulfilled by Docker containers, settings, and services running within these containers. The use of Terraform as a platform allowed these containers to be deployed easily from a single script, with configurations working as designed. Combined with proper version control, a containerised infrastructure created in this manner can be securely updated, tested, and deployed safely. An added bonus is the ability to destroy and recreate the environment exactly as designed if configurations or settings change from compliant settings.

4.2.2. Unsuccessful implementation of requirements

Of the twelve high-level PCI-DSS requirements used as assessment criteria, the implementation of a containerised infrastructure was deemed to have not explicitly met one of them:

- 3. Protect stored cardholder data.

This can be considered a critical failure in the implementation of a containerised infrastructure. Having sensitive data exposed in anyway is major breach of PCI-DSS compliance in general. This was cause by an issue in the storage method, Docker volumes. While the data within can be encrypted in MySQL Server itself, the fact that by default Docker volumes are accessible by a user with 'root' privileges should not be ignored. This could be mitigated by encrypting the VM and its virtual hard disk on the Hyper-V host, but within the VM itself there is still the risk of data being compromised. If considering containers to replace or work with existing infrastructure, to meet this PCI-DSS requirement it may be more prudent to use a different storage method. Even if Docker volume was encrypted, it persists only on the Docker host. If a container and its network are moved to another server, unless that server is configured correctly for failover with all data copied then the containers will be unable to access the Docker volume and its data.

4.2.3. Untestable requirements under current conditions and implementation

Of the twelve high-level PCI-DSS requirements used as assessment criteria, this implementation was unable to be assessed against four of them:

- 4. Encrypt transmission of cardholder data across open, public networks.
- 8. Identify and authenticate access to system components.
- 9. Restrict physical access to cardholder data.
- 12. Maintain a policy that addresses information security for all personnel.

These four requirements being considered untestable under current conditions raised questions about the test environment itself, and what would actually be required to assess a containerised infrastructure against them to meet PCI-DSS requirements. While the

successful and unsuccessful requirements can be assessed in a test environment, these four have characteristics that would likely require a production environment to adequately test. This production environment would most likely be a business considering the use of containers and automation for their services. Such a business would be able to robustly test user account permissions, interaction with the public internet, physical security, and implement and effective information security policy.

4.3. Results interpretation

The results of the assessment of the project implementation against high-level PCI-DSS requirements, and their categorisation, would suggest that the technologies used in this project are capable of being compliant with PCI-DSS. The major exception to this would be how Docker handles its storage, which by default is insecure and reliant on the Docker host. While not in the majority, the number of requirements unable to be tested in the current environment suggests that in order properly assess anything against PCI-DSS requirements, more extensive resources and a reasonable approximation of a production environment would be required.

Results being considered ‘successful’ or ‘unsuccessful’ should not be considered definite when assessing PCI-DSS requirements, rather they should be used as as an assessment of the viability of such an idea, and if it is worth pursuing. Although this project focuses on the twelve high-level requirements, when examined in detail the number of components and criteria for each of the twelve is extensive. To examine a single requirement and all its criteria fully could arguably make for entire project in itself.

4.4. Chapter summary

This chapter detailed the results of a qualitative assessment of the implemented containerised infrastructure against the twelve high-level PCI-DSS compliance requirements. Results showed that, in general, the technologies selected were appropriate in meeting compliance standards with the major exception of storage. Several requirements revealed issues with the test environment, and to what extent this might need to be changed in order to adequately assess a containerised infrastructure against PCI-DSS requirements.

5. Conclusions

Throughout the history PCI-DSS compliance, it has been proven to be an extensive and effective compliance standard. From governmental inquiries to market analyses, it has been verified as effective in securely storing credit card information. So effective is the standard that it has seen adoption by businesses and organisations outside of the payment card industry. Its twelve high-level requirements provide the basis for an organisation to assess their compliance against, and from there drill down into specific components and criteria. When examined at a high-level, the requirements appear as relatively simple, common sense objectives that any organisation should implement. However as past cases and investigations have shown, these requirements are only effective if they are correctly implemented. Partial implementation is simply not enough to protect against data breaches, as has been demonstrated time and time again. A particularly noteworthy point is that of all the investigations and audits into business that claimed to PCI-DSS compliant, so far none that was fully compliant has ever been breached. It is for this reason that PCI-DSS compliance requirements were selected for use in this project. The individual components and criteria of the high-level requirements are extensive, and while beyond the scope of this project they are important to be aware of, as even if something successfully meets all twelve high-level requirements, it could still fail at meeting individual criteria. It could be argued that the details of each requirement are worth their own individual projects. For the purpose of determining just how involved adhering to compliance standards is, the high-level requirements proved effective. Considering this project involved an extremely simple test environment and still faced difficulties, a new appreciation for the work required in achieving and remaining compliant has been developed. This is an appreciation that can be utilised in industry application, as no matter what the business does it is highly likely they will have information systems and services that must be kept up to compliance standards. It can reasonably assumed the adherence to every component of PCI-DSS is a massive task, and a project likely to involve a team of individuals with the expertise to make this goal a reality. Regardless of the difficulties involved, ensuring the technology and information used by organisations is kept secure is critical, and PCI-DSS provides a well-proven and time-

tested method of doing so. It is increasingly important to examine new and developing technologies against these requirements, as despite their relative newness, leveraging technologies such as containerisation can have massive benefits to cost and performance for a business.

Virtualisation has become a fact of life for modern businesses, and as technology continues to advance the need for it grows. Virtual machines have become well-established, and their use in industry is the current standard. A common usage of virtual machines is running legacy applications which are no longer compatible with modern hardware or operating systems. This requires spinning up a virtual machine that is effectively an entire operating system, just to run one or more applications. This is an incredibly wasteful use of resources, but it is still necessary as businesses and organisations continue to rely on legacy applications. The development of container platforms such as Docker addressed this resource usage, providing a way to run an application with only its dependencies while sharing the host operating system kernel. This method is resource effective, provides better performance, and scales well. A feature of containers is their ability to be destroyed and redeployed very quickly, which translates to quicker development times, making them a suitable technology for modern DevOps processes. However these processes are subject to their own information security and best practice standards, and a measured approach should be required for the adoption of any new technology. In implementing PCI-DSS standards, the use of containers and their specific requirements need to be thoroughly examined before anything is put in to production. As the results of this project have shown, meeting compliance requirements is no simple task, even for an uncomplicated setup.

The results found that in general, the technologies that were used could be considered suitable for the aims of this project. However, a major stumbling block from beginning to end was the quality of documentation available. While Docker is an established technology, much of the documentation and available resources are still primarily concerned with older versions with legacy features. These features, while functional, are discouraged from use by official Docker documentation. Terraform is a much more recent technology than Docker, and as such its documentation is comparatively lacking. Official documentation can be incomplete or bare bones, and during the design phase of this project, many solutions and problems with syntax and implementation were solved thanks to user forums and other

unofficial support. This would be acceptable for learning purposes, but in considering a technology for use in meeting PCI-DSS, higher standards should be expected. A lack of official documentation and references leaves room for ambiguity. Without clear cut examples of best practice to base designs from, this could leave a Terraform-deployed infrastructure vulnerable. While there is no doubt that Terraform is a powerful and useful orchestration tool, it is still being developed, and using a similar but more well-established technology may be a better choice if aiming for full PCI-DSS compliance. There is also the fact that Terraform itself is primarily an orchestration tool, and if used solely for this purpose it is likely to be used in a compliant manner. Pairing it with a suitable configuration management platform that can also meet compliance requirements should be serious consideration for any organisation attempting to introduce automation.

The inconclusive results for several PCI-DSS requirements can be used to consider what constitutes an acceptable test environment. The available resources involved in the test environment could be considered extremely limited, in both computing power and the location testing was carried out in. It is possible several of these could be addressed, with hypothetical plans for physical security and user management, but without an environment in which to adequately test these plans the results would still be inconclusive. Considering changes to the design, the transit of data over public networks could be tested if another test machine was available. Apart from this, any plans for information security and user management would require at least some testing to gauge their effectiveness in meeting PCI-DSS compliance.

The one objective failure of this implementation, securing sensitive data, needs to be addressed further. Docker volumes are by default not secured, and even with their contents encrypted by the applications running in the containers that use them, they are still dependent on the Docker host. If aiming to containerise the storage of sensitive information, Docker host security needs to be prioritised or another solution used. Even if the volume is secured appropriately, implementing failover or other forms of disaster recovery could be a difficult task, as the storage is bound to that Docker host specifically. It could be argued that securely storing data is the primary concern of PCI-DSS, and failure to meet this requirement renders success in other areas inconsequential.

As discussed throughout this paper, there were significant issues with configuring the firewall container, with the end result functionally the same as if a firewall service was just configured on the Docker host. This result, along with the failure caused by the Docker volume, greatly influenced the conclusions of this project. To address the research question, which seeks to assess a the viability of containerised infrastructure in meeting PCI-DSS requirements, the answer is not a simple matter of 'yes' or 'no'. As the results have shown, the technology is capable of meeting high-level PCI-DSS requirements, but can be problematic to get working as intended. In the case of the firewall, considerable research was needed to achieve the same result as simply installing a firewall package on the Docker host. However the other containers running MySQL Server and phpMyAdmin were relatively simple, and are functionally identical to those services running on their own virtual machines. This leads to the conclusion that containers are a viable option to consider when aiming for PCI-DSS compliance, but with several caveats. As the results have shown there is much ambiguity that needs to be resolved, and the specific criteria of those high-level requirements that containers succeed in meeting need to be investigated further. A hybrid infrastructure consisting of containers and virtual machines providing services could be a way to achieve full PCI-DSS compliance. Containerising those services which can be verified as compliant, and function with performance near to the same applications on bare metal, and using virtual machines for those that do not, could lead to a successful hybrid infrastructure. The advantage of using an orchestration tool such as Terraform means that this environment can be scripted much the same way as a wholly containerised environment, and be subject to the same development processes and testing as a containerised infrastructure deployed via automation.

Appendices

Appendix A: References

Alpine Linux, 2020 Alpine Linux (2020). About. Available: https://alpinelinux.org/about/. Last accessed 10th Jul 2020.

Bui, 2015 Bui, T., 2015. Analysis of docker security. arXiv preprint arXiv:1501.02967.

Canonical, 2020 Canonical (2020). Basic Ubuntu Security Guide, Desktop Edition. Available: https://wiki.ubuntu.com/BasicSecurity#Acknowledgements. Last accessed 10th Jul 2020.

Catuogno and Galdi, 2016 L. Catuogno and C. Galdi, "On the Evaluation of Security Properties of Containerized Systems," 2016 15th International Conference on Ubiquitous Computing and Communications and 2016 International Symposium on Cyberspace and Security (IUCC-CSS), Granada, 2016, pp. 69-76, doi: 10.1109/IUCC-CSS.2016.018.

Chen et al, 2015 Chien-Ming Chen, Shaui-Min Chen, Wei-Chih Ting, Chi-Yi Kao, and Hung-Min Sun, "An enhancement of return address stack for security", Computer Standards & Interfaces, Volume 38, 2015, Pages 17-24, ISSN 0920-5489, https://doi.org/10.1016/j.csi.2014.08.008.

Cito et al, 2016 J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi and H. C. Gall, "An Empirical Analysis of the Docker Container Ecosystem on GitHub", 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), Buenos Aires, 2017, pp. 323-333, doi: 10.1109/MSR.2017.67.

Combe et al, 2016 T. Combe, A. Martin and R. Di Pietro, "To Docker or Not to Docker: A Security Perspective," in IEEE Cloud Computing, vol. 3, no. 5, pp. 54-62, Sept.-Oct. 2016, doi: 10.1109/MCC.2016.100.

Docker, 2020a Docker (2020). Docker overview. Available: https://docs.docker.com/get-started/overview/. Last accessed 10th Jul 2020.

Docker, 2020b Docker (2020). Use volumes. Available: https://docs.docker.com/storage/volumes/. Last accessed 10th Jul 2020.

Docker, 2020c Docker (2020). Use bind mounts. Available: https://docs.docker.com/storage/bind-mounts/. Last accessed 10th Jul 2020.

Docker, 2020d Docker (2020). Use bridge networks. Available: https://docs.docker.com/network/bridge/. Last accessed 10th Jul 2020.

Docker, 2020e Docker (2020). Legacy container links. Available: https://docs.docker.com/network/links/. Last accessed 10th Jul 2020.

Docker, 2020f — Docker (2020). Docker and iptables. Available: https://docs.docker.com/network/iptables/. Last accessed 10th Jul 2020.

Drutskoy et al, 2013 — D. Drutskoy, E. Keller and J. Rexford, "Scalable Network Virtualization in Software-Defined Networks," in IEEE Internet Computing, vol. 17, no. 2, pp. 20-27, March-April 2013, doi: 10.1109/MIC.2012.144.

HashiCorp, 2020a — HashiCorp (2020). Introduction to Terraform. Available: https://www.terraform.io/intro/index.html. Last accessed 10th Jul 2020.

HashiCorp, 2020b — HashiCorp (2020). Configuration Language. Available: https://www.terraform.io/docs/configuration/index.html. Last accessed 10th Jul 2020.

HashiCorp, 2020c — HashiCorp (2020). Docker Provider. Available: https://www.terraform.io/docs/providers/docker/index.html. Last accessed 10th Jul 2020.

HashiCorp, 2020d — HashiCorp (2020). Terraform Recommended Practices. Available: https://www.terraform.io/docs/cloud/guides/recommended-practices/index.html. Last accessed 10th Jul 2020.

Galabov, 2019 — Galabov, V (2020). Red Hat's container software strategy paying off, for now. Available: https://technology.informa.com/617145/red-hats-container-software-strategy-paying-off-for-now. Last accessed 10th Jul 2020.

Göktas et al, 2018 — E. Göktas et al., "Position-Independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure," 2018 IEEE European Symposium on Security and Privacy (EuroS&P), London, 2018, pp. 227-242, doi: 10.1109/EuroSP.2018.00024.

Kang et al, 2016 — H. Kang, M. Le and S. Tao (2016), Container and Microservice Driven Design for Cloud Infrastructure DevOps, 2016 IEEE International Conference on Cloud Engineering (IC2E), Berlin, 2016, pp. 202-211, doi: 10.1109/IC2E.2016.26.

Kreutz et al, 2014 — Kreutz, D., Ramos, F.M., Verissimo, P.E., Rothenberg, C.E., Azodolmolky, S. and Uhlig, S., 2014. Software-defined networking: A comprehensive survey. Proceedings of the IEEE, 103(1), pp.14-76.

Li et al, 2017 — Z. Li, M. Kihl, Q. Lu and J. A. Andersson, "Performance Overhead Comparison between Hypervisor and Container Based Virtualization," 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA), Taipei, 2017, pp. 955-962, doi: 10.1109/AINA.2017.79.

Leite et al, 2020 — LEITE, L. et al. (2020) 'A Survey of DevOps Concepts and Challenges', ACM Computing Surveys, 52(6), pp. 1–35. doi: 10.1145/3359981.

Maheshwari et al, 2018 — Maheshwari, S., Deochake, S., De, R. and Grover, A., 2018. Comparative Study of Virtual Machines and Containers for DevOps Developers. arXiv preprint arXiv:1808.08192.

Martin et al, 2018 — Martin, A. et al. (2018) 'Docker ecosystem – Vulnerability Analysis', Computer Communications, 122, pp. 30–43. doi: 10.1016/j.comcom.2018.03.011.

Microsoft, 2014 Microsoft (2014). Secure Boot. Available: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-8.1-and-8/dn486875(v=ws.11)?redirectedfrom=MSDN. Last accessed 10th Jul 2020.

Microsoft, 2016a Microsoft (2016). Create a virtual switch for Hyper-V virtual machines. Available: https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/get-started/create-a-virtual-switch-for-hyper-v-virtual-machines. Last accessed 10th Jul 2020.

Microsoft, 2016b Microsoft (2016). Should I create a generation 1 or 2 virtual machine in Hyper-V?. Available: https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/plan/should-i-create-a-generation-1-or-2-virtual-machine-in-hyper-v. Last accessed 10th Jul 2020.

Moldes, 2018 Moldes, C. (2018). Compliant but not Secure: Why PCI-Certified Companies Are Being Breached. Available: https://www.csiac.org/journal-article/compliant-but-not-secure-why-pci-certified-companies-are-being-breached/. (Accessed 23 May 2020).

Oracle, 2020a Oracle (2020). 12.13 Encryption and Compression Functions. Available: https://dev.mysql.com/doc/refman/8.0/en/encryption-functions.html#function_encrypt. Last accessed 10th Jul 2020.

Oracle, 2020b Oracle (2020). 6.1 General Security Issues. Available: https://dev.mysql.com/doc/refman/8.0/en/general-security-issues.html. Last accessed 10th Jul 2020.

PCI-SSC, 2018a Cloud Special Interest Group PCI Security Standards Council (2018). Information Supplement: PCI SSC Clous Computing Guidelines. Available: https://www.pcisecuritystandards.org/pdfs/PCI_SSC_Cloud_Guidelines_v3.pdf. (Accessed: 24 May 2020).

PCI-SSC, 2018b Payment Card Industry (PCI) (2018). Data Security Standard, Requirements and Security Assessment Procedures Version 3.2.1. Available: https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-2-1.pdf?agreement=true&time=1590279327741. (Accessed: 24 May 2020).

phpMyAdmin, 2020 phpMyAdmin (2020). About. Available: https://www.phpmyadmin.net/. Last accessed 10th Jul 2020.

Rahman et al, 2019 Rahman, A., Farhana, E., Parnin, C. and Williams, L., 2019. Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts.

Red Hat, Inc., 2012 Red Hat, Inc. (2012). Position Independent Executables (PIE). Available: https://access.redhat.com/blogs/766093/posts/1975793. Last accessed 10th Jul 2020.

Rogers and Lyman, 2016 Rogers, O., Lyman, J., 451 Research LLC. (2016). Containers economically, they appear to be a better option than hardware virtualization (Accessed: 24 May 2020).

Seo et al, 2014 Seo, K.T., Hwang, H.S., Moon, I.Y., Kwon, O.Y. and Kim, B.J., (2014). Performance comparison analysis of linux container and virtual machine for building cloud. Advanced Science and Technology Letters, 66(105-111), p.2.

Suo et al, 2018 K. Suo, Y. Zhao, W. Chen and J. Rao, "An Analysis and Empirical Study of Container Networks," IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, Honolulu, HI, 2018, pp. 189-197, doi: 10.1109/INFOCOM.2018.8485865.

Sysdig, 2019 Carter, E., Sysdig, Inc. (2019). Sysdig 2019 Container Usage Report: New Kubernetes and security insights. Available: https://sysdig.com/blog/sysdig-2019-container-usage-report/. (Accessed: 24 May 2020).

Tosatto et al, 2015 A. Tosatto, P. Ruiu and A. Attanasio, "Container-Based Orchestration in Cloud: State of the Art and Challenges," 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems, Blumenau, 2015, pp. 70-75, doi: 10.1109/CISIS.2015.35.

US GPO, 2010 US Government Printing Office – Congressional Hearings (2010), Do the Payment Card Industry Data Standards Reduce Cybercrime? Hearing Before the Subcommittee on Emerging Threats, Cybersecurity, and Science and Technology of the Committee on Homeland Security, House of Representatives, One Hundred Eleventh Congress, First Session, March 31, 2009. Available: https://www.hsdl.org/?view&did=27800. (Accessed 23 May 2020).

Veritas, 2019 Veritas Group, Inc. (2019). A Whitepaper on Containerization. Available: https://www.veritis.com/wp-content/uploads/2019/09/whitepaper-container-technology-services-tools-veritis-overview-1.pdf. (Accessed: 24 May 2020).

Verizon, 2015 Verizon (2015). Verizon 2015 PCI Compliance Report (Executive Summary). Available: https://enterprise.verizon.com/resources/reports/pci-report-2015-executive-summary.pdf. (Accessed: 23 May 2020).

Verizon, 2019 Verizon (2019). 2019 Payment Security Report. Available: https://enterprise.verizon.com/resources/reports/2019-payment-security-fullreport-bl.pdf. (Accessed: 23 May 2020).

Wright, 2009 Wright, S. (2009) PCI DSS : A Practical Guide to Implementation, Second Edition. Ely, UK: IT Governance Ltd. Available at: http://search.ebscohost.com/login.aspx?direct=true&AuthType=sso&db=e000tww&AN=391106&site=ehost-live (Accessed: 23 May 2020).

Xingtao et al, 2018 L. Xingtao, G. Yantao, W. Wei, Z. Sanyou and L. Jiliang, "Network virtualization by using software-defined networking controller based Docker," 2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference, Chongqing, 2016, pp. 1112-1115, doi: 10.1109/ITNEC.2016.7560537.

Zamora et al, 2019 Martín Zamora, P. et al. (2019) 'Increasing Windows security by hardening PC configurations', EPJ Web of Conferences, 214, p. 1. Available at: http://search.ebscohost.com/login.aspx?direct=true&AuthType=sso&db=edb&AN=139062212&site=eds-live&scope=site (Accessed: 10 July 2020).

Appendix B: Links to required resources

Lubuntu 20.04 image: https://lubuntu.me/downloads/

MySQL Docker image: https://hub.docker.com/r/mysql/mysql-server/

iptables Docker image: https://hub.docker.com/r/paulczar/iptables

phpMyAdmin Docker image: https://hub.docker.com/r/phpmyadmin/phpmyadmin

'ddeploy.tf' Terraform script:
https://github.com/danecode/lyit-terraform-project/blob/master/ddeploy.tf

'systeminfo' output:
https://github.com/danecode/lyit-terraform-project/blob/master/systeminfo_full.txt

'terraform plan' output:
https://github.com/danecode/lyit-terraform-project/blob/master/terraform_plan.txt

'terraform apply' output:https://github.com/danecode/lyit-terraform-project/blob/master/terraform_apply.txt

'daemon.json' file:
https://github.com/danecode/lyit-terraform-project/blob/master/daemon.json

Appendix C: Evidence of successful deployment and configuration

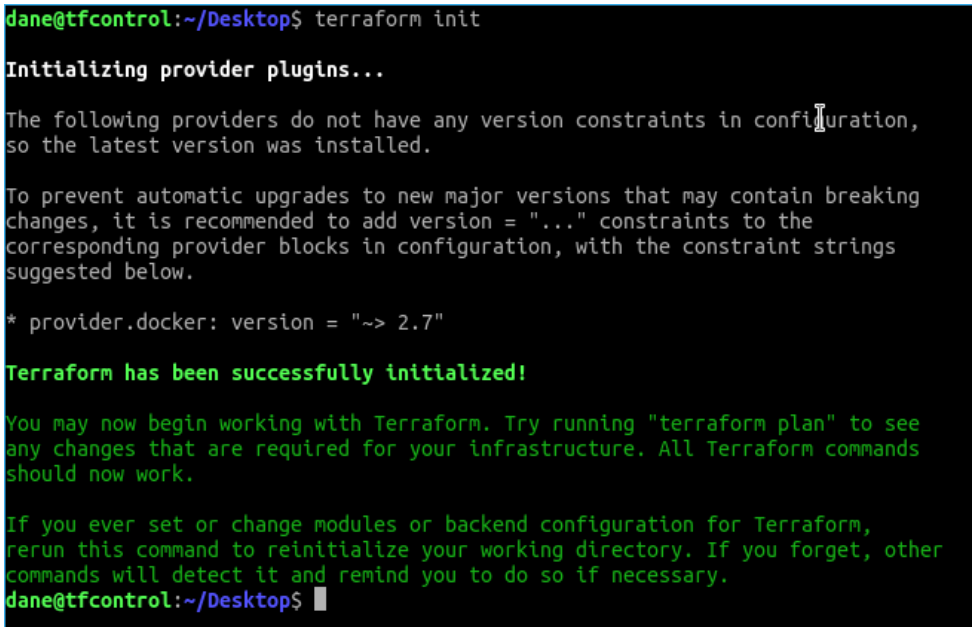

Figure 3: 'terraform init' output

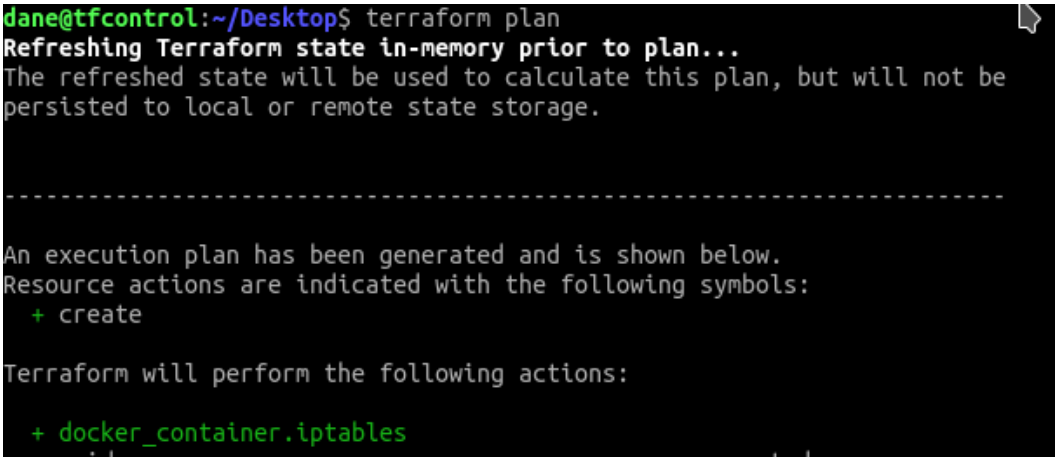

Figure 4: 'terraform plan' partial output 1

```
+ docker_image.iptables
    id:                                     <computed>
    latest:                                 <computed>
    name:                                   "paulczar/iptables:master"

+ docker_image.mysql-server
    id:                                     <computed>
    latest:                                 <computed>
    name:                                   "mysql-server:latest"

+ docker_image.phpmyadmin
    id:                                     <computed>
    latest:                                 <computed>
    name:                                   "phpmyadmin:latest"

+ docker_network.private_network
    id:                                     <computed>
    driver:                                 "bridge"
    internal:                               <computed>
    ipam_config.#:                          "1"
    ipam_config.375909234.gateway:          ""
    ipam_config.375909234.ip_range:         ""
    ipam_config.375909234.subnet:           "172.20.0.0/16"
    ipam_driver:                            "default"
    name:                                   "dbnet"
    options.%:                              "1"
    options.com.docker.network.bridge.name: "dbnet"
    scope:                                  <computed>
```

Figure 5: 'terraform plan' partial output 2

```
Plan: 7 to add, 0 to change, 0 to destroy.

------------------------------------------------------------------------

Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.

dane@tfcontrol:~/Desktop$
```

Figure 6: 'terraform plan' partial output 3

```
Plan: 7 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

docker_network.private_network: Creating...
  driver:                                 "" => "bridge"
  internal:                               "" => "<computed>"
  ipam_config.#:                          "" => "1"
  ipam_config.375909234.gateway:          "" => ""
  ipam_config.375909234.ip_range:         "" => ""
  ipam_config.375909234.subnet:           "" => "172.20.0.0/16"
  ipam_driver:                            "" => "default"
  name:                                   "" => "dbnet"
  options.%:                              "" => "1"
  options.com.docker.network.bridge.name: "" => "dbnet"
  scope:                                  "" => "<computed>"
docker_image.phpmyadmin: Creating...
  latest: "" => "<computed>"
  name:   "" => "phpmyadmin:latest"
docker_image.mysql-server: Creating...
  latest: "" => "<computed>"
  name:   "" => "mysql-server:latest"
docker_image.iptables: Creating...
  latest: "" => "<computed>"
  name:   "" => "paulczar/iptables:master"
docker_network.private_network: Creation complete after 3s (ID:
docker_image.iptables: Still creating... (10s elapsed)
```

Figure 7: 'terraform apply' partial output1

```
Apply complete! Resources: 7 added, 0 changed, 0 destroyed.
dane@tfcontrol:~/Desktop$
```

Figure 8: 'terraform apply' partial output 2

```
dane@tfcontrol:~$ sudo docker ps --format="$FORMAT"
ID      164ae5582dbb
NAME    mysql_server
IMAGE   716286be47c6
PORTS   0.0.0.0:3306->3306/tcp, 33060/tcp
COMMAND "/entrypoint.sh mysq…"
CREATED 2020-07-09 23:28:37 +0100 IST
STATUS  Up 15 minutes (healthy)

ID      eb168e4f36fe
NAME    php_admin
IMAGE   6f9550cff175
PORTS   80/tcp, 0.0.0.0:80->8080/tcp
COMMAND "/docker-entrypoint.…"
CREATED 2020-07-09 23:25:24 +0100 IST
STATUS  Up 19 minutes

ID      28ed554f1979
NAME    firewall
IMAGE   135b2c3b39e2
PORTS
COMMAND "/bin/ash 'iptables …"
CREATED 2020-07-09 23:21:12 +0100 IST
STATUS  Restarting (2) 25 seconds ago

dane@tfcontrol:~$
```

Figure 9: Docker containers running after deployment

```
dane@tfcontrol:~$ sudo docker exec -it firewall /bin/ash
/ # iptables -S
-P INPUT DROP
-P FORWARD DROP
-P OUTPUT ACCEPT
-N DOCKER
-N DOCKER-ISOLATION-STAGE-1
-N DOCKER-ISOLATION-STAGE-2
-N DOCKER-USER
-A INPUT -i lo -j ACCEPT
-A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
-A INPUT -p tcp -m tcp --dport 8080 -m state --state NEW,ESTABLISHED -j ACCEPT
-A INPUT -p tcp -m tcp --dport 3306 -m state --state NEW,ESTABLISHED -j ACCEPT
-A INPUT -s 172.20.0.0/16 -p tcp -m state --state NEW,ESTABLISHED -j ACCEPT
-A INPUT -i dbnet -j ACCEPT
/ #
```

Figure 10: iptable firewall rules inside 'firewall' container

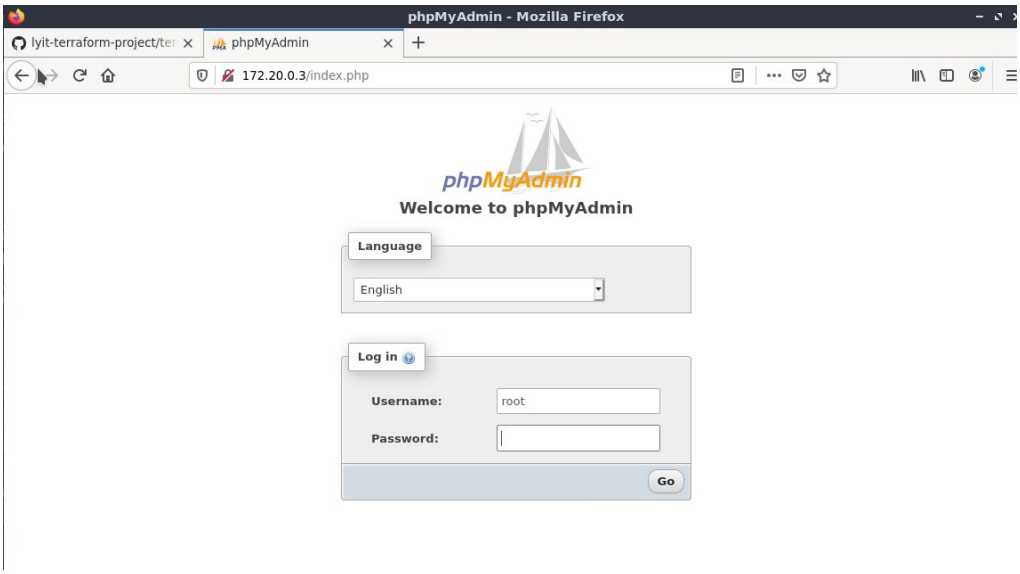

Figure 11: phpMyAdmin login screen

Figure 12: phpMyAdmin main page post-login

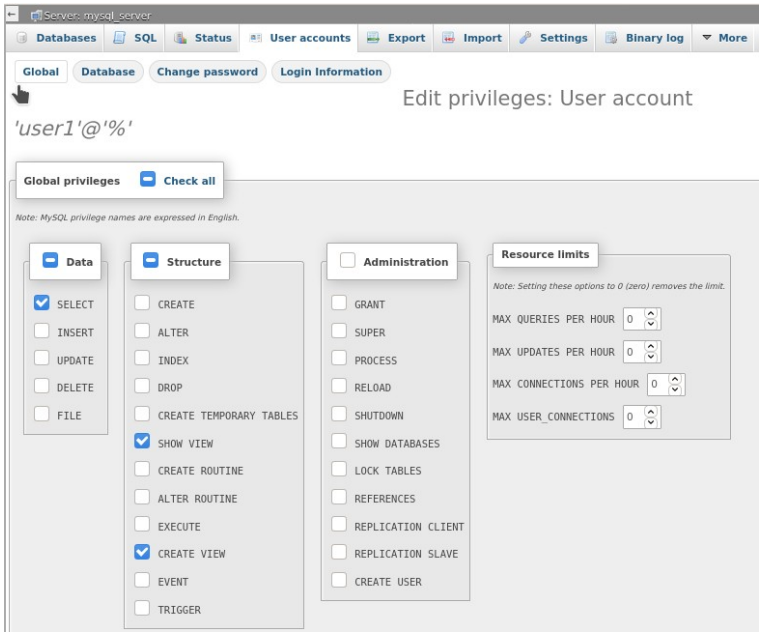

Figure 13: Altering 'user1' permissions

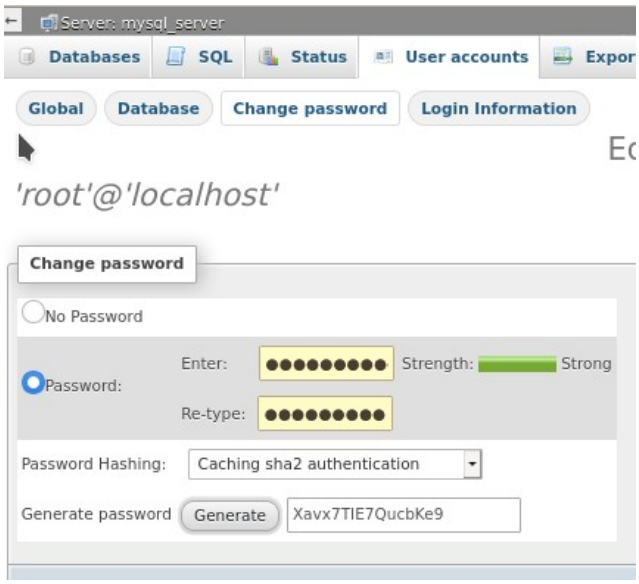

Figure 14: modifying 'root' MySQL password

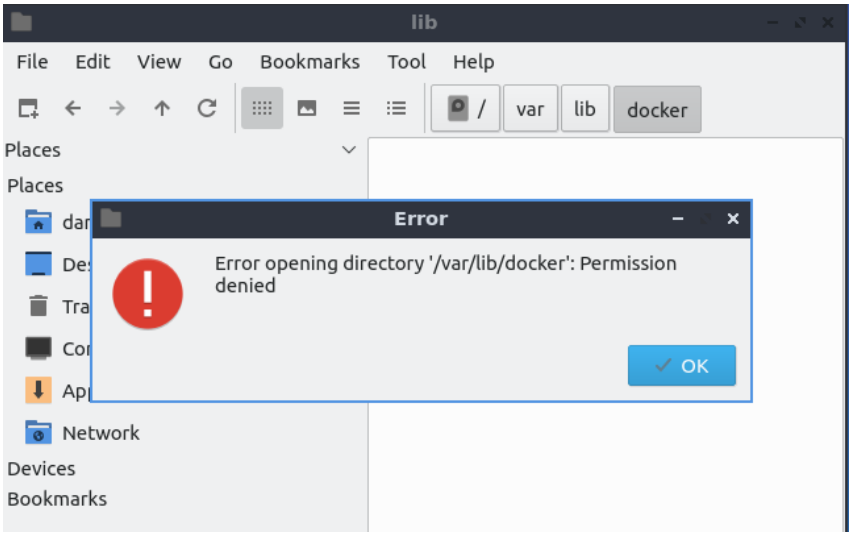

Figure 15: trying to access Docker volume as normal user

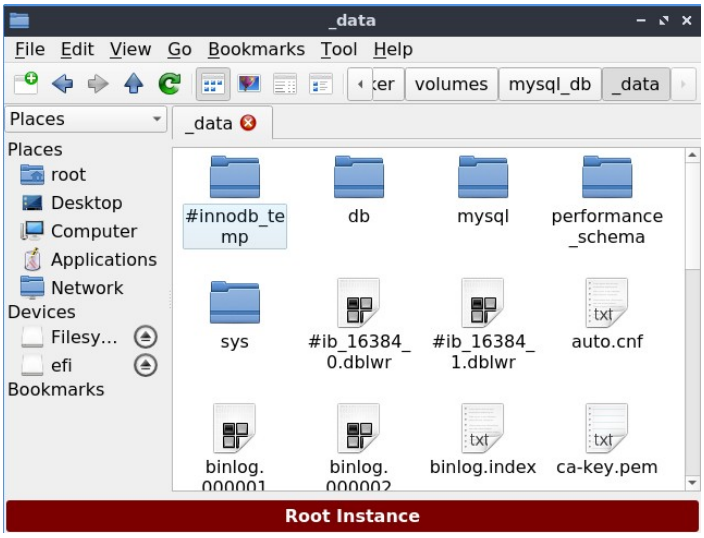

Figure 16: Accessing Docker volume with 'root' privileges