M Ű E G Y E T E M 1 7 8 2

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Management and Vulnerability Scanning of Docker Containers in an Embedded Environment

Bachelor's Thesis

Author
Zsolt László Czikó

Advisor
dr. Levente Buttyán
Dorottya Futóné Papp

December 12, 2019

Contents

Kivonat **i**

Abstract **ii**

1 Introduction **1**

2 Related-work **3**

2.1 IoT security 3

2.2 Generally about containerization 4

2.2.1 Containerization vs virtualization 4

2.3 Docker ecosystem 5

3 Working with more containers **6**

3.1 Orchestrators 6

3.1.1 Comparing orchestrators 7

3.1.2 Docker Swarm 8

4 Use cases for Docker in IoT **11**

4.1 Small sized environment 11

4.2 Middle sized environment 12

4.3 Huge sized environment 13

5 Tools for managing Docker containers 15

5.1 About managing containers 15

6 Tools for vulnerability scanning of Docker containers 19

6.1 About vulnerability scanning of containers 19

6.1.1 Docker Healthcheck 21

6.1.2 Analysis of the security tools 21

7 Conclusion 26

Acknowledgements 28

Bibliography 29

HALLGATÓI NYILATKOZAT

Alulírott *Czikó Zsolt László*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2019. december 12.

Czikó Zsolt László
hallgató

Kivonat

Napjainkban majdnem minden elektronikus eszköz csatlakozik az internethez, így a beágyazott eszközök is. Ezt nevezzük IoT (Internet of Things)-nek. Ezek viszonylag olcsó eszközök, nagyon szigorú erőforrás megkötésekkel. Az IoT eszközöket olcsó áruk és széles alkalmazási lehetőségeik miatt az iparban nagy mennyiségben használják. De többségük nincs teljesen kihasználva. Továbbá jelenleg sem biztonságosak, és a jövőben a számuk növekedésével egyre nagyobb biztonsági kockázatot fognak jelenteni. Valamint az irányítása ennyire sok eszköznek még egy meg nem oldott probléma.

Ebben a dolgozatban ajánlok egy megoldást az imént említett problémákra. A megoldásom alapja egy létező technológia használata, amelyet gyakran alkalmaznak hagyományos eszközökön. Ez a konténerizáció orkesztrációval, de beágyazott eszközökön. Bemutatok felhasználásokat, amik bizonyítják, hogy a konténerizáció használata beágyazott eszközön szükséges. Ezek után összehasonlítok már létező eszközöket, amik minden egyes felhasználáshoz segítenek irányítani a beágyazott környezetet. Végezetül részletezem a biztonsággal kapcsolatos problémákat, és megoldást kínálok azokra már létező biztonsági eszközök használatával. A célom az eszközökkel, hogy megvizsgáljam vajon képesek -e futni egy beágyazott eszközön, valamint hogy mennyire terhelik meg a rendszert.

Abstract

These days almost every electronic device is connected to the internet, thus are the embedded devices too. This is called IoT (Internet of Things). They are relatively cheap devices, with very strict resource limitations. They are being used in the industry in large quantities, because of their low price and their widespread applicability. But most of them are not completely utilized. Furthermore, they are not secure at the present time, and in the future with the growth of the number of these devices, they will imply more and more security risks. As well as, the management of so many devices is not a solved problem right now.

In this paper, I propose a solution for all of the short while ago mentioned problems. The base of my solution is the usage of an already existing technology, which is often used on traditional devices. This is containerization with orchestration, but I work on embedded devices. I provide use-cases, which prove that the usage of containerization on embedded devices is necessary. After that I compare already existing tools, that help to manage an environment in each use-case. Finally I detail the problems around security, and I offer a solution for those with the usage of already existing security tools. My goal with the tools is to analyse whether they are able to run on an embedded device, and how much do they charge the system.

Chapter 1

Introduction

The Internet of Things (as known as IoT) means a lot of devices connected to the internet about to ensnare the world. This ambition is a good thing on the one hand, because it aims to make life easier and more comfortable, but on the other hand, this includes a lot of danger, defencelessness. The principle of these devices is to collect and send data about the users, about people to analyse those for important informations.

The appearance of the IoT devices influenced the industry positively. These devices can be manufactured in large quantities for very low prices. That is because, they use embedded processors, sensors, which has small performance, but for the aim of the usage it is completely enough.

People love using IoT devices, because they make their life easier. Nowadays, every single electronic device is smart. It began with smartphones, smart watches, and now even the fridges are smart. This technology carried a lot of advantages in almost every area.

Furthermore, not only are production costs low, the upkeep of these devices costs nearly nothing. Because of the architecture, they consume very little electricity, and replacement parts are available for low prices as well.

The exact number of the IoT devices is unknown, but nowadays it is spreading everywhere. And this spreading will not decrease in the future, more and more cities will become smart, by 2020 the number of these devices can reach 5.8 billion.[1] And with this really huge number comes a lot of problems.

The management of this many devices is insoluble with the currently used technology, because the maintenance of them one by one will take forever. Not to mention the fact that the basic idea was to use them to save resources, and a lot of them will not be probably

[1]https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io

fully utilised. The aim is to run as many applications as possible on one embedded device, because with this attitude one can save energy, money, and avoid the existence of the idling devices. In summary: automated management of the devices and applications is necessary in the future, and the currently existing security problems will get worse, so it is the other key-question. [14]

These are not unknown problems, and in the world of non-embedded devices the solution is the usage of virtualization technology. But an embedded device cannot run a virtual machine, because of the physical resource limits. Here comes container based virtualization (as known as containerization) in picture. Containerization technology provides nearly as much as traditional virtualization technology, but with much less overhead.

Nowadays, containerization is a traditional solution to run multiple applications in separate way. However, this technology is not tested on embedded devices, even though it can provide the solution for all of IoT devices problems. The biggest name in this area is Docker. Docker ecosystem has a lot of complementary tools to handle the above mentioned problems.

The rest of the thesis is structured as follows. Chapter 2 presents the background for this thesis, including challenges of IoT devices, containerization and the Docker ecosystem. Then in Chapter 3, there are ways to solve the problem of managing a lot of containers. Chapter 4 discusses the use-cases of Docker in IoT environments. Then in Chapter 5, there are tools that help manage the system, and I give recommendations of the tools for each use-cases. Chapter 6 presents tools that scan containers for vulnerabilities, to defend the systems against attacks, and I give recommendations of the tools for each use-cases too. Finally Chapter 7 concludes the thesis.

Chapter 2

Related-work

2.1 IoT security

In the world of IoT every resource matters. Embedded devices have limited size of memory(RAM and Storage too), and processing performance, because they have to be cheap. These little, low-priced devices are all around us, in every smart solution. The problem comes with the lack of security: they surround us, know (almost) everything about us, and they do not have a proper implemented security system, or they do not have at all, so during an attack the attacker can get all information about us. Security solutions cost not so much, but the usage of them will rise the expenses at least a little, and can make the deployment slower.

The lack of defence causes a lot of problems nowadays, and this will get worse in the future with the increase of the number of embedded devices. Let's take a look for a few attacks that recently happened: "The malware family behind 39% of attacks - Mirai - is capable of using exploits, meaning that these botnets can slip through old, unpatched vulnerabilities to the device and control it. Another technique is password brute-forcing, which is the chosen method of the second most widespread malware family in the list – Nyadrop. Nyadrop was seen in 38.57% of attacks and often serves as a Mirai downloader. This family has been trending as one of the most active threats for a couple of years now. The third most common botnet threatening smart devices - Gafgyt with 2.12% - also uses brute-forcing."[1]

To minimise the risk of a security failure the devices should use security solutions. There are 3 different approaches for security: preventive, reactive and detective. I will detail them more in Chapter 6, but for now the important thing is to protect the system with

[1]https://www.kaspersky.com/about/press-releases/2019_iot-under-fire-kaspersky-detects-more-than-100-million-attacks-on-smart-devices-in-h1-2019

the least overhead. Detective tools are often used in traditional environments, but the continuous system scanning costs a lot of resources, which embedded devices do not have. The usage of preventive and reactive tools can provide a secure base for the system without big overhead. But the development of a new tool costs very much time, and money too.

There are a lot of already existing tools for containerization security, so if embedded devices start to use containerization, then the question of the missing tools disappears. However, containerization is not tested on embedded environment, but the tests do not cost so much as developing a new security tool. To begin the work let's give a look at the basics of containerization.

2.2 Generally about containerization

Containerization provides a virtualized environment of an operating system(only the application layer is virtualized). It can be considered as a lightweight virtual machine. It packages up the software and all of its dependencies, so it can run equally on every platform. A container is made up of a base image (which can be ubuntu, debian, busybox, etc.), and the application, and all of its joint configs, libraries, and other dependencies have to be added to the container. Containerization ensures portability, that the application can work in diverse computing environments, and it ensures isolation from other containers, and from the host system too(but keep in mind, that containers use the same kernel as the host, that is why it is lightweight).

2.2.1 Containerization vs virtualization

In the past, if one wanted to run different applications separate, then the only way was to use a hypervisor, which could create and run virtual machines. This was a good solution, because the system administrators could adjust the available physical resources of the machines, and handle them one-by-one. This was better, than using another physical machine for each separate task. The hypervisor creates separate kernels for each virtual machine, which cost a lot of wasted physical resources. But with the growing demand on virtual machines, this size of waste is unacceptable.

Containerization, also known as container-based virtualization provides isolation, and portability at least the same way, as traditional virtualization does, but in most cases better. Linux Containers(LXC) use the kernel of the host system, so does not waste resources to build, and run a new one for the container, and it is still be able to run multiple copies in a separate way.[26] With the usage of a containerization technology (eg. Docker or CoreOS) the construction of the images can be shortened, which can reduce deployment time. [20][24]

All in all, containerization is a better way to run multiple separate applications. However, one must not forget that it uses the same kernel for each container, so it is not as secure as traditional virtualization. But in an embedded environment, where one has little amount of physical resources(often less than 1 GB of RAM, etc.) containerization is able to provide isolation for processes with minimal resource overhead.

2.3 Docker ecosystem

There are a few containerization technologies (e. g. CRI-O, rktlet, Microsoft Containers), but Docker [7] is the pioneer of containerization, it was the first, that offered open-source solution. One might say Docker is a de facto standard in container based virtualization. Furthermore, it is the most used container based technology all over the world, and it has a really good documentation, so the realization of it is simple, and fast. Besides there are a lot of existing tools that can supplement Docker's work. [12]

Docker has some default services, for example `docker-compose`, which is a tool to create more containers at once. Another useful service is docker-hub: it is an online database, where one can store their images, which can then be deployed from there. The usage of hub is free until a certain size (1 private repository, with 1 build), but for a moderate price this can be increased. Moreover, there are a lot of already existing images on hub, from which, one can start to develop their own image. Accordingly Docker does not occupy a lot of space, because it downloads only the needed base images, and it checks at every launch, if there are any changes in the image: it has a built in automatic update service.

But there are cases where the default services are not enough: here comes supplementary tools in picture. These can help with orchestration, continuous integration/deployment, monitoring, logging, security, storage/volume management, networking, service discovery, building an image, management of a containerized system. These tools are necessary to make Docker(and the system) better, safer and more easily controlable.

The architectural base of a containerized system is the image. First one has to make a Dockerfile, which defines the base image, and where one can add its application to its image (and all of the dependencies of its application). After that one has to build this image from the Dockerfile. Now one can run containers based on its image. This order is really important, because it protects the containers from departure failures (if the Dockerfile is faulty, then the image will not be created), and it insures an opportunity to analyze the container (the image will be analyzed) for vulnerabilities before deploying it. One best practice is to analyze the system before it goes live, and this method is absolutely accessible with containerization. The difference between containers and images is that containers are running images. One can have more than one containers from the same image.

Chapter 3

Working with more containers

As detailed in Chapter 2, applications should run in containers, because containerization provides better security, and isolates the application from the host system and from each other. Furthermore, to offer a service, more than one applications are needed, and it is not practical to run every application on different devices. To utilize the physical resources of IoT devices, they should run as many applications (containers) as possible. But working with more containers is not very easy: deploying them, and the load balancing of the application can be a very complex problem. The solution of these problems on Desktop was the usage of an orchestrator.

3.1 Orchestrators

"Orchestration is a pretty loosely defined term. It is broadly the process of automated configuration, coordination, and management of services. In the Docker world we use it to describe the set of practices around managing applications running in multiple Docker containers and potentially across multiple Docker hosts." [1]

An orchestrator does the load balancing (equipartition between the containers) , scaling(if all of the applications are occupied, then it starts new application(s)) instead of us. To do this it has to manage the containers: start, stop, and restart them, scale them.

The difference between the regular solutions and solutions for embedded devices comes from the attributes of the embedded devices. The lack of resources makes the work of the orchestrators very hard.

[1]Extract from James Turnbull, The Docker Book: Containerization is the new virtualization book

A system with an orchestrator should have at least one master node. The master node monitors the rate of the charging of the system, and it makes a decision about the scaling. The default case is where the master node does not work (it does not run any copy of the application), it works only with the load balancing, and scaling. The applications run on worker nodes, which are controlled by the master node. [22]

The architectural design of the system depends on the number of applications (containers). With the increase of the number of applications grow the complexity of the system, in parallel with that grow the necessity of an overall management tool. But this will be discussed in Chapter 4.

Without orchestrators the environment cannot react alone to the load, and cannot scale itself: the environment needs somebody to pay attention to it, and it is really expensive. So orchestrators are necessary for keeping the costs low, and for the continuity of the service.

3.1.1 Comparing orchestrators

Nowadays, 3 big orchestrators can be discussed: Docker Compose, Kubernetes, Docker Swarm. Before Kubernetes got incomparable among the orchestrators there were a lot more, but they slowly collapsed (because of the lack of interest, and money), and Kubernetes became leader.

- Docker Compose:

  "Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration."[2] So Docker Compose is a "container" for containers. It is not a classical orchestrator like Kubernetes, or Swarm, because it cannot do the load balancing, and cannot auto scale the application. This is just a tool, to start more than one container at the same time, and it is used for testing.

- Kubernetes:

  "Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery."[3]

  Kubernetes is the biggest, and most popular container orchestrator of the world. I thought about working with Kubernetes, but the free to use version is not enough for big sized environments, and the resource demand of Kubernetes is very high,

[2]https://docs.docker.com/compose/
[3]https://kubernetes.io/

cannot operate well with embedded devices. So Kubernetes is out of scope for this project.

3.1.2 Docker Swarm

"Docker Swarm or simply Swarm is an open-source container orchestration platform and is the native clustering engine for and by Docker."[4]

Swarm is the biggest challenger of Kubernetes, because it can do all the things, that Kubernetes can, and swarm can deploy containers faster then Kubernetes, and it is totally free to use. The manager node(s) deploy the applications as services on the worker nodes. When deploying a new service one can set the number of replicas, the permissions of the container to the host system, etc. The smallest unit of the swarm is task. A task is made up of a container, and the commands to run inside the container. Specifically a master node assigns a task to a worker node, not a service. The number of replicas of the service is based on the scaling, a worker node can run more than one replica(tasks) of the same application. Global services are running with one replica, but they have to run on every node(worker and manager too) in the cluster. Global services will be very important to understand how Manager tools work (discussed in Chapter 5).[16]

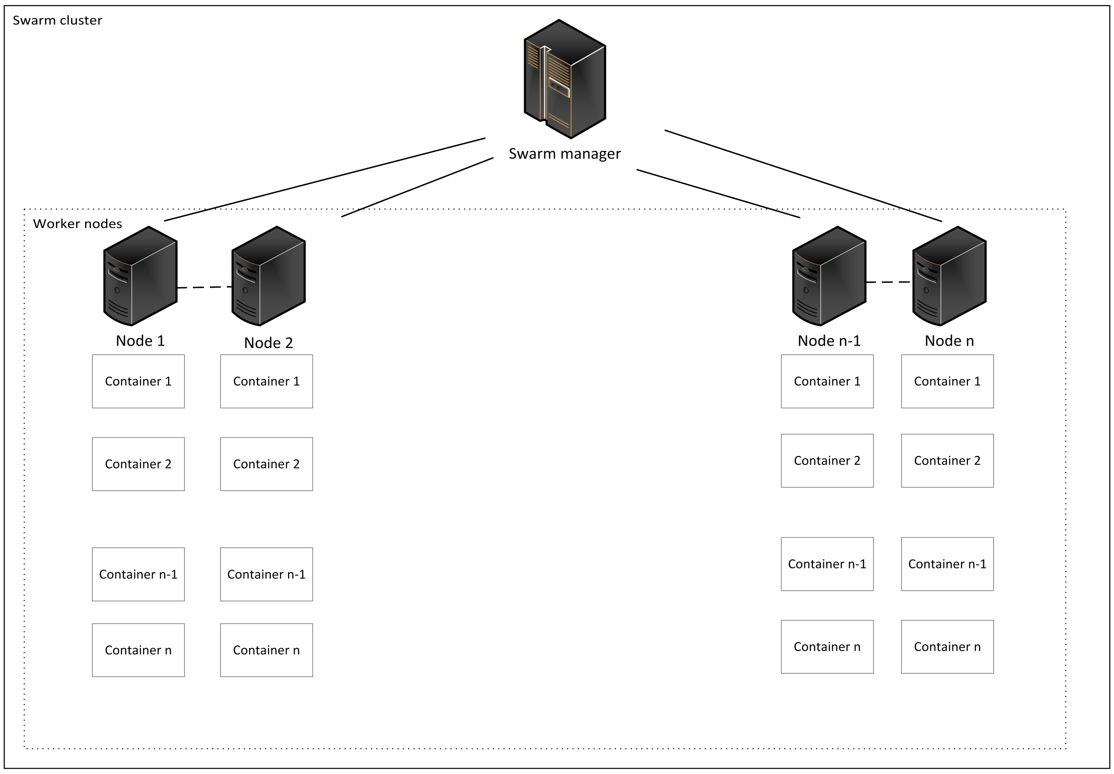


Figure 3.1: The architectural design of a swarm

[4]https://thenewstack.io/kubernetes-vs-docker-swarm-whats-the-difference/

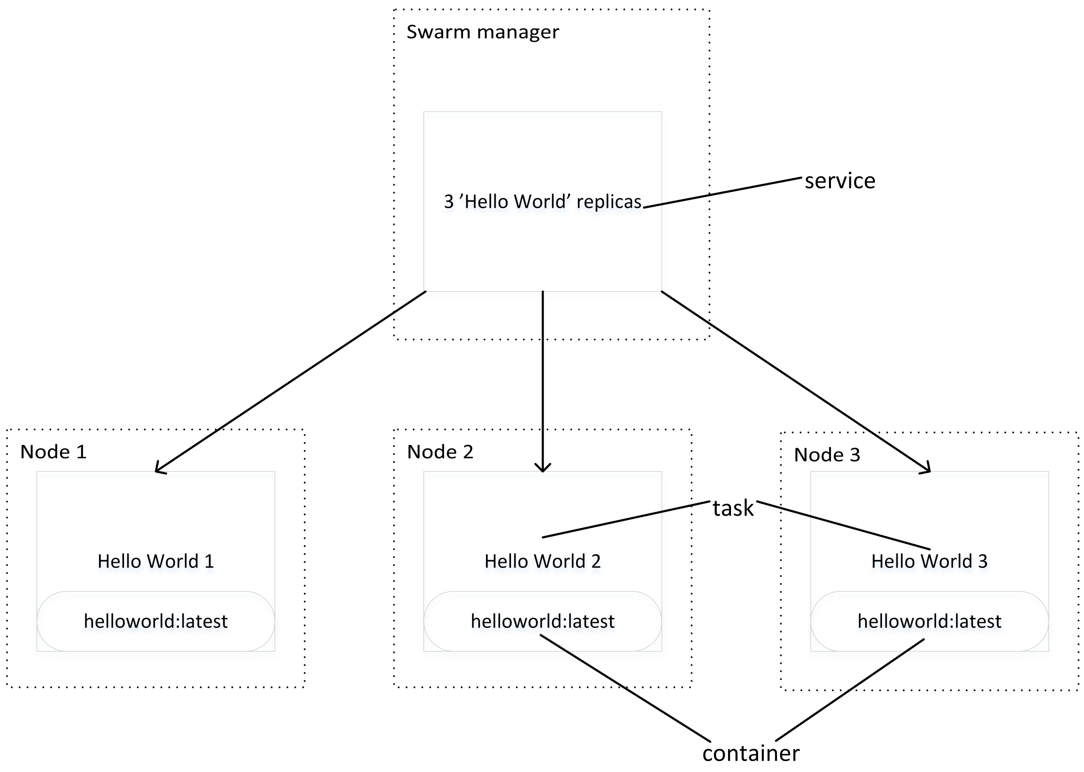


Figure 3.2: Connection between containers, services and tasks

It is easy to deploy, because it uses Docker Engine, so one does not need to install another application/service. It is possible, to connect a new host anytime, whether a manager node, or a worker node. The automated scaling, load balancing, and continuous monitoring provides a reliable service: if the charging is too huge, swarm can make new replicas of the application, and if a replica fails, then it creates another one instead of the wrong one. It is a secure solution, because it uses by default TLS mutual authentication and encryption between the nodes, and of course it is possible to use any self-signed certificate to make it more secure. It supports rolling updates: if the new version of the application does not work like one expected, one can easily roll back to the last working version.

The working of swarm is simple: the future manager node creates the swarm, and it creates a join token, which has to be used on the other hosts to join the swarm. All the host machines have to run in swarm mode: this means that Docker Engine version has to be 1.12 or later.

The last reason beside swarm is that, it works well with manager tools: the short while ago mentioned global functions make the manager tools deployment simple, because if one wants to manage the containers, one has to run a copy of the manager application on every host, and that is the global function by definition.

On figure 3.1 you can see the typical architecture of a Swarm (with one master node, which manage a lot of worker nodes, which have lots of containers). Figure 3.2 shows the connections between containers, services and tasks. Although it is not depicted in the figure, every node can have more than one containers. For the reasons given above I decided to use Docker Swarm as the project's Orchestrator.

Chapter 4

Use cases for Docker in IoT

I sketched a use-case before, where one has a lot of containers, and a lot of host machines. But the reality could be very different. The size of the company determine the number of hosts, and it could move on a wide range. In my opinion there are 3 different scenarios: small, medium, and big sized environments. The architecture of the containerized system will be different in every case, based on economical and practical decisions. The environment need to be controlled by the orchestrator, and this makes the use-cases very different: the number of the manager nodes, and the host who runs it.

4.1 Small sized environment

As Figure 4.1 shows, in this case there are only 1-2 host machines (embedded devices) controlled. For this size I do not recommend using any dedicated controller, because the upkeep of the controller can be bigger than the advantage it brings. Instead of a dedicated controller the developer can use his own computer to handle the containers(the embedded devices), and to analyze them. If one offers a service, and the customer just uses it, then one can oversee the hosts(the containers, the swarm) on one's system, there is no need for a dedicated device. This works only for small environments, where the load is not huge, so the manager roles are reduced to security scan the images, and to check the availability of the containers (load balancing and scaling do not play role).

This can be good for small, self made smart home systems, where the user only has a few sensors (for example: temperature sensor, huditimity sensor) and they want to manage automatically the equipment of the house (for example: the heating, the shutters, the lamps).

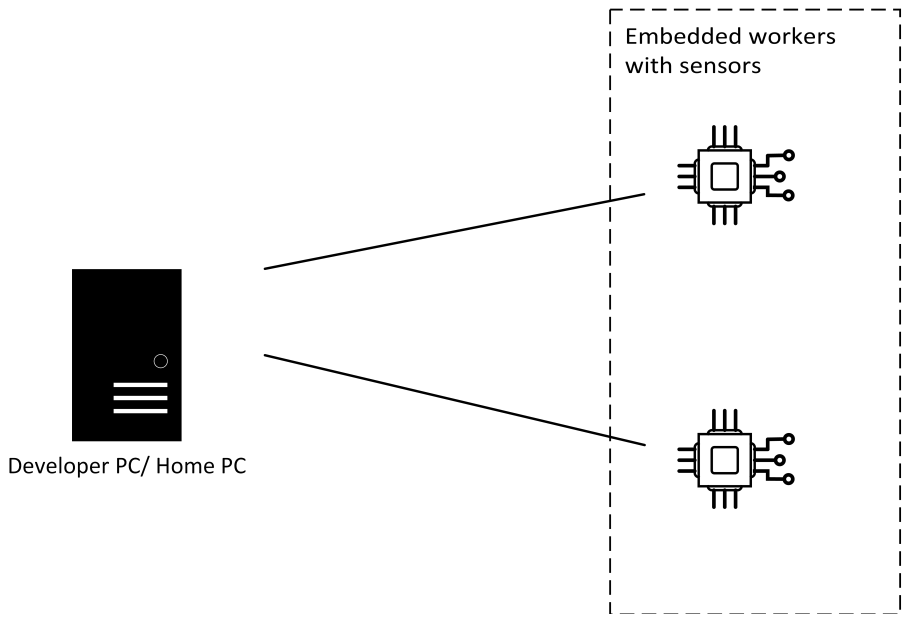


Figure 4.1: Example for small sized environment architecture

4.2 Middle sized environment

The difference between medium and small environment comes from the number of the devices(the number of the applications): in this case one has to use more than 2 devices, because one wants to run a few applications (even the same application). Controlling, configuring, and updating (in short: managing) this number of devices one by one cannot be done without any kind of orchestration tool, because it will take too long, and it will not be practical. I recommend using a dedicated controller, which can be an embedded device too. If there are not so many devices, than one contoller should be enough, but for a bit more worker hosts, I would use at least two controllers: it is great, because the correction of the malfunctions can be done simultaneously, and because this can help to manage the load (for example at a Denial of Service attack). Furthermore it is good because of redundancy: if one of the controllers fails, than the system keeps working.

This use-case is perfect for small shops or small sized factories. In the preceding case one does not have to count on big turnover, so one does not have to be afraid of running out of capacity. For the latter, a few devices can manage the manufacturing processes without any interventions.

The limitation of the system comes from the embedded device controller. Although it is a cheap solution, it has serious limits: it can easily run out of processing performance, and memory. With the increase of the number of controllers the process can be delayed, but it cannot be terminated. So for bigger environments other solutions are required.

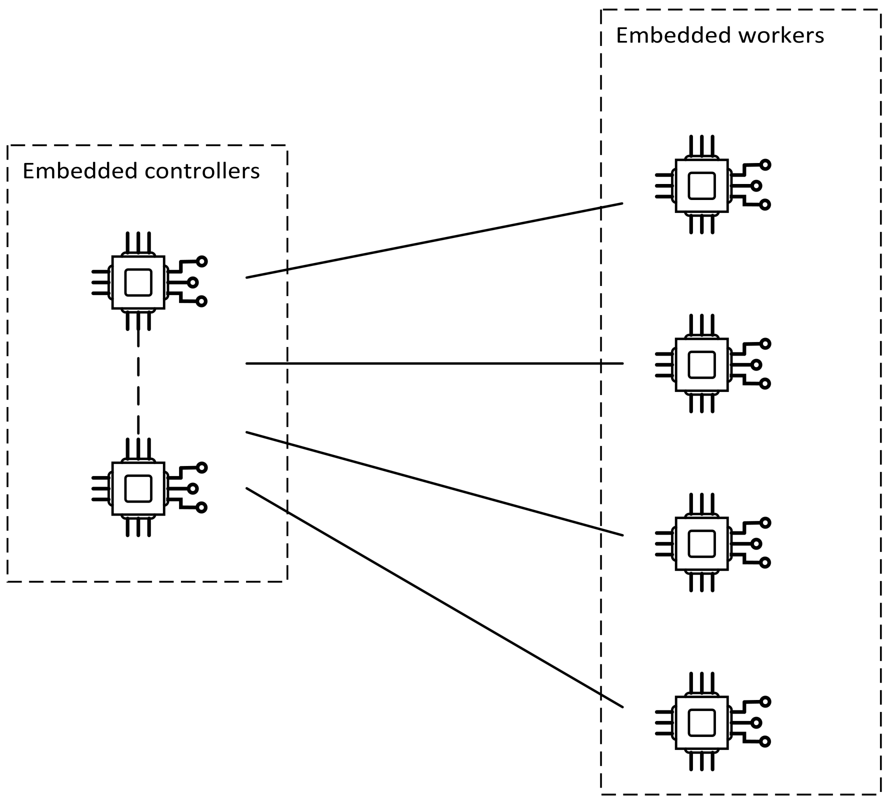


Figure 4.2: Example for medium sized environment architecture

4.3 Huge sized environment

In this case the number of host machines(embedded devices) can reach a very big number (from 50 to almost unlimited), so the problem can be very complex. The problems of middle sized environments also applies here: configuring and updating of hundreds of devices cannot be done one by one, an overall management tool is necessary (this can be any kind of orchestration technology). Until a certain size, one dedicated, non embedded controller can handle the orchestration tasks. But for a huge sized system one controller is not enough: this does not seem like a big problem, but the architecture of the controllers is not obvious. But they are not embedded devices, so the traditional solutions are good for them. Otherwise, I recommend to use at least 2 controllers (even if it is not needed), because redundancy gives the system more reliability: if suddenly the load increases (for example at a Denial of service attack), then it would not be a problem, because the second controller can help with the load balancing, and scaling tasks, and if one of the controllers fails, then the other one can take over the lead of the swarm.

This can work for big factories: the production can be monitored and managed by embedded devices. If additional services are provided, than the limitations of the system depend only on the number of embedded devices. The enlargement of the number of host machines is simple, and relatively cheap. This makes the system flexible, and easily scalable.

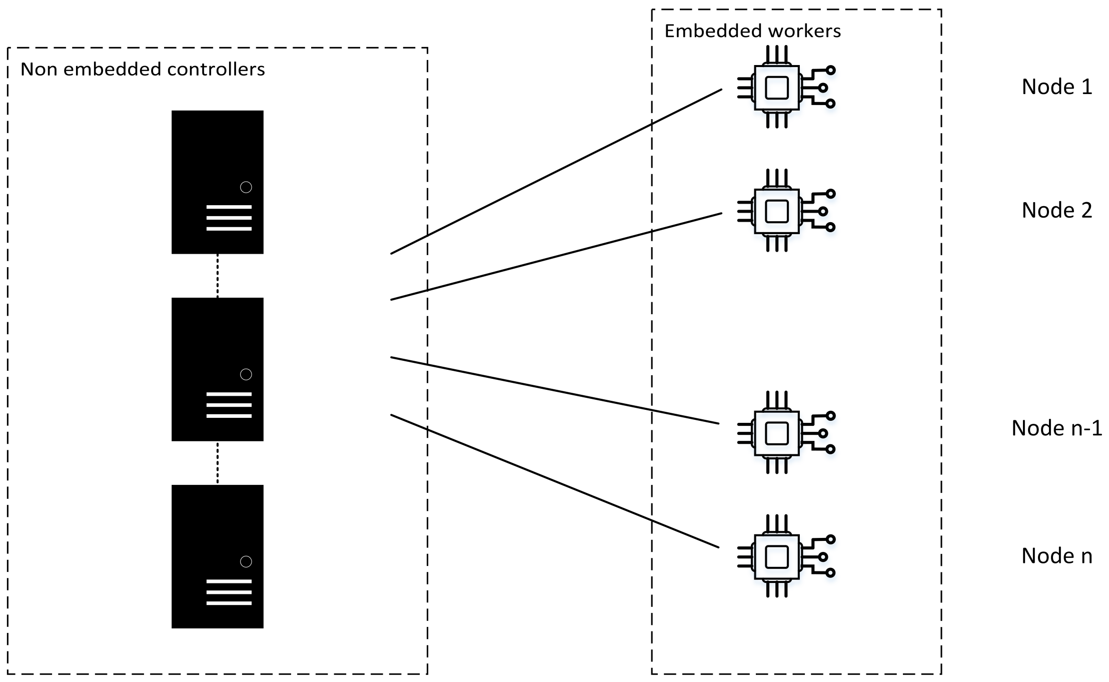


Figure 4.3: Example for huge sized environment architecture

The number of IoT devices nowadays is near 4.8 billion, and in the future this number will rise rapidly. Without proper planning they will become untreatable. The security of the devices are already questionable (for example Botnet attacks [13][2]), and with this fast increase this will not change. For all these reasons, I think the usage of some containerization tool is necessary: Docker is the most popular container based virtualization technology, with the most complementary tools, but any other could be a good choice too.

Chapter 5

Tools for managing Docker containers

With more containers comes a new problem: management without tools becomes impenetrable, inconvenient. As seen before Docker Swarm can do the automated management of the containers, but it cannot show an overall picture about the system, and it cannot do the expansion of the host machines.

5.1 About managing containers

A Docker container can be in 3 different states: created (that is called Image), running, stopped. The lifecycle management of a container is very important to keep the application running and the service stable. For each use-case -discussed in Chapter 4.- the management structure can be diverse: the priorities of a personal smart home are different from the priorities of a big factory. There are a lot of solutions for this problem for regular PC's, but the embedded ecosystem has serious limitations, so the compatibility is not obvious.

The operational principle of a management tool is to run a container on every host machine, so the manager can reach the worker nodes. This brings up questions about resource management in an embedded ecosystem. Certainly running an extra container is not good from the view of the load, but the transparency of the system with a management tool is much better than without it. The developer of the system has to consider what is more important: the performance of the nodes, or manageability of the environment. The strengths and weaknesses of the previously discussed use-cases can aid in the decision.

Managing containers on a few hosts are very different from managing a swarm. Swarm is not so wide-spread, so relatively few tools can handle it. Without built-in swarm support the deployment of the manager tool is difficult: the nodes has to be added to the manager node one-by-one. This is a bit inconvenient, but it eliminates the extra load of running an extra container on every host. Most of the tools do not need to be installed, they can run as a docker container, so they can manage themselves, and they can work well with the already available technologies. This makes the requirements of the tools low, which is nice from the view of development.

I tested 4 tools on embedded devices (on Raspberry PI 3 Model B), these were: Portainer[15][19], DockStation[8], Docker Compose UI[25], Swarmpit[23]. They are all frequently used on traditional platform to manage containers. Table 5.1 shows the comparison of the tools. My main viewpoints were:

- GUI:

  One of the most important things in a management tool is the GUI(graphical user interface): the containers can be managed without a tool for that, but this makes the system chaotic. A good, logical interface can simplify the work.

- Installation and maintenance:

  The difficulty of deploying can easily determine the destiny of a tool. Basically I distinguish 2 categories: first, where installation is needed, and second, where the tool can run as a docker container. From the view of the maintenance both ways can be good, but the containerized tool can be updated easily with the restart of the container, because at the starting of the container it has to update to the latest version (found on Docker Hub).

- Support:

  The new features, bug fixes, the documentation: they are playing a serious role in the life of a service. The management tools are services, which show a simpler picture about the system, and make the management easier. If something goes wrong, or something does not work like it used to, then maybe you can find the answer in a good (or existing at all) documentation.

- Identifying swarm:

  If the tool cannot identify swarm, and despite this all of the nodes has been connected, one cannot get an overall review of the system as a unity: one can just see the load of a/more node(s), but not all of them (except there are only a few nodes). This viewpoint with the resource demand can be crucial.

- Resource demand:

  There is a close coherence between the resource demand and the usage of swarm. If the tool can handle Docker Swarm, then it creates a replica on every host, and the supervision of the node is done. This implies surplus load, which can be decisive in an embedded environment. In contrast to this -without swarm- the manager tool has to connect to the nodes somehow(it can be done via SSH, or by an opened port on the host). The connection can be done one-by-one, so this is not a very convenient way, and for a lot of hosts it is nearly impossible.

- Communication with Docker:

  The base question is, how does the tool get informations about the containers, about the swarm.

In the previous chapter I presented 3 different use-cases. In my opinion, the usage of Swarm(or any kind of orchestrator) is necessary for medium and big sized environment. In these scenarios, I recommend using a tool, what can identify Swarm to maximize the usefulness. Portainer or Swarmpit can be a good choice. For small sized environments -where are only a few hosts- DockStation can be a good option, because connect a few nodes to the manager node one-by-one is not so painful, as well as in this case there are not any surplus load on the workers, which is important for a small sized environment.

Let's take a look for a use-case, which was not mentioned up to now: testing a new application/service. The key to the testing is the simplicity, the easy deployment. Docker Compose UI is not good to analyze a complex system, but it makes the deployment of a group of containers easier, and simpler. With this tool the reading of logs become easy, and this can speed up the development/testing.

	Portainer	DockStation	Docker Compose UI	Swarmpit
GUI	Can show charts about the whole system; understandable surfaces; can be accessed via browser on the port one has granted; remote hosts console can be accessed in the browser	Modern, good looking, simple; nice charts; separate application	Just a web interface for Docker Compose (to deploy new containers); does not have charts; can be accessed via browser	Modern, has a lot of diagrams, charts, mobile-friendly GUI; can be accessed via browser on port 888
Installation and maintenance	Can deployed as a container	Installation needed: not every platform has support; strict requirements for the supported platforms	Can deployed as a container	Can deployed as a container
Support	Detailed documentation; frequent updates, bug fixes; big community	Does not have documentation; rare updates; moderate size community	Does not have documentation; the project looks abandoned: no updates since May 2018; small community	Semi-detailed documentation; frequent updates, bug fixes; big community
Identifying swarm	Yes	No	No	Yes
Resource demand	Runs a container on every node of the swarm, this means surplus load	Because it does not run in a container, and cannot use swarm, it does not make surplus load on the hosts (only on the manager node), so it has nearly zero resource demand.	To access remote hosts, the host has to run a Docker Compose UI container, so it makes a surplus load on the hosts.	Runs a container on every node of the swarm, this means surplus load
Communication with Docker	It uses Docker API	It uses Docker API	It uses Docker API	It uses Docker API

Table 5.1: Comparison of management tools

Chapter 6

Tools for vulnerability scanning of Docker containers

As I mentioned before, the security of the embedded devices today are really neglected. This causes a lot of problems (for example: the mirai botnet, hackable cardiac devices, observation through cameras/webcameras[1]) and in the future if people will not pay attention to that, then this will get worse.

6.1 About vulnerability scanning of containers

Containerization gives an extra security layer to the system with the separate running from the host, and from other containers. The permissions (access to the host system, the internet, etc.) of the container can be specified. But the basics of the containerization is to use the same kernel as the host system. This raises a lot of questions: how strong is the separation between the container and the host? To run a container the user has to be part of the Docker group, which is almost equal to root privileges. An exploit in the container (outbreak to the host) can give the attacker nearly root privileges. [3][4]

These are known issues in non embedded environments: security tools, and other measures should take care for the security of the system. The realization of the best practices are not obvious on embedded devices: in this environment - where every resource matters - it is not allowed to waste any. This will supply the basis of my analysis.

There are 3 different approaches for security:

- Preventative:

[1]https://www.iotforall.com/5-worst-iot-hacking-vulnerabilities/

In this case one would like to minimize the security risks, and the potential impact of a successful threat event. The best practices include policies, standards, procedures, encryptions, firewalls, secure boot.

- Detective:

  The principle of detective controls is to detect a just happening attack, or to notice the attack as soon as possible. This includes static analysis of the logs, network intrusion detection and identification of malicious code.

- Reactive:

  When trouble happened one would like to get rid of it as soon as possible, here come reactive controls in picture. The most important thing is to recover to a stable state, this can be done by a rollback(to follow a recovery plan), or by removal of malicious code.

Preventative methods can work well with embedded devices, because they do not consume (usually) extra resources. Static analysis of images is a great way to protect the system: one can test the image with the application inside against well known security issues, and for common best-practices around Docker containers. This type of security approach fits well in the use-cases, because the deploying of the images should be done by the manager node(s), and before the deploy one can make a static analysis of the image on the manager node easily.

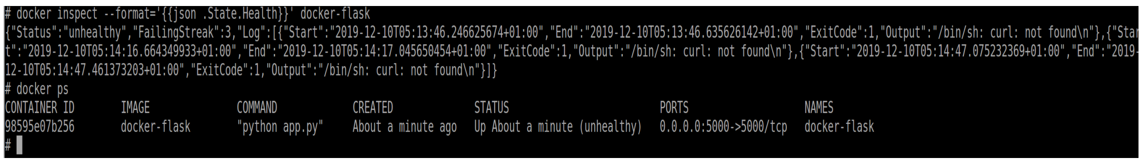
Detecting a currently humming attack would be nice, but on embedded devices this almost cannot come into consideration: the scanning of the containers has a very big resource demand (even 20% of the system resources). The only way to get information without high overhead is to analyze the logs of the containers: this cannot give enough information, but still better than nothing. Fortunately Docker has a built in method to analyze if the container is running correctly: that is called Healthcheck. [6]

Assume that an attack has been detected against the system. The solution for this with containers is very easy: restart the containers, or stop them. This means temporary spillage in the service, but with orchestrators this can be done quickly, so it minimizes the damage. The only problem occurs, when the attacker can break out from the container, and obtain the control over the embedded device, but the treatment of this scenario belongs to another theme.

Knowing all these, I decided to work with preventative tools. But before the compare let's discuss about Docker Healthcheck.

6.1.1 Docker Healthcheck

The reason behind this analysis is that Docker comes with this tool built-in (Docker version >1.12). This is not a security tool in the classical sense. The detection of the health-state of a container depends in large from the developer. It is because healthcheck does not use any kind of default database, the developer can write a command, which will be executed in the container. If this command terminates successfully, then Docker stigmatizes the container as healthy. At the deploy, man can give an interval, and the timeout time, and the number of retries: the interval means how frequent the healthcheck will be done. If a check takes longer than the timeout, Docker considers it as a failed check. If a check fails, it does not mean that the container is unhealthy: thereupon is the number of retries important. If all of them fails, then Docker stigmatizes the container as unhealthy (Figure 6.1 shows an example output), otherwise as healthy. This functioning does not certify that the container is healthy, only if the executing command cannot be terminated during an attack.



```
# docker inspect --format='{{json .State.Health}}' docker-flask
{"Status":"unhealthy","FailingStreak":3,"Log":[{"Start":"2019-12-10T05:13:46.246625674+01:00","End":"2019-12-10T05:13:46.635626142+01:00","ExitCode":1,"Output":"/bin/sh: curl: not found\n"},{"Star
t":"2019-12-10T05:14:16.664349933+01:00","End":"2019-12-10T05:14:17.045650454+01:00","ExitCode":1,"Output":"/bin/sh: curl: not found\n"},{"Start":"2019-12-10T05:14:47.075232369+01:00","End":"2019-
12-10T05:14:47.461373203+01:00","ExitCode":1,"Output":"/bin/sh: curl: not found\n"}]}
# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED              STATUS                         PORTS                    NAMES
98595e07b256        docker-flask        "python app.py"     About a minute ago   Up About a minute (unhealthy)  0.0.0.0:5000->5000/tcp   docker-flask
# 
```

Figure 6.1: Example for unhealthy container

For example: the application is a web server, and the command is to reach the main page of the site. If the system is under a DoS attack, then the check will fail in all likelihood. This case healthcheck was useful, because it detected the problem in the service. But if the system is facing against an attack, that is not block the run of the server (e.g. a spyware attack), then healthcheck will not catch the problem.

Furthermore the continuous testing means a lot of surplus load. Then why do not use a detective tool instead of healthcheck. Detective tools cannot insure complete defense against attacks too, and they have to be installed on the hosts. In contrast to this, healthcheck came with Docker automatically.

As it can be seen, there are advantages and disadvantages beside the usage of healthcheck, and in some cases I would consider using this tool.

6.1.2 Analysis of the security tools

Nowadays more and more companies started to use containerization, so the security of a containerized application is very important: because of this there are a lot of security tools. Apart from the above mentioned categories, there is another viewpoint to look at them: there are commercial and open-source applications to defend the containers.

- Commercial security tools:

  Although the commercial tools did not constitute part of the project(because it focuses on free to use tools, and solutions), I undeliberately noticed some tools which showed up in every research/study. These were Aqua MicroScanner [21], BlackDuck Docker Security [9] and HashiCorp Vault [11]. The two preceding are good for static analysis of images, and the latter is a tool for managing secrets (e.g. passwords, certificates, access tokens), which is important in an environment where copies alternate frequently.

- Open-source security tools:

  I tested 5 security tools on embedded devices(on Raspberry PI 3 Model B), these were: Anchore Engine [1], CoreOS Clair [18], Dagda [5], Docker-bench Security [10], Notary [17]. The reason behind my selection is the very high number of mentions of these tools in other researches/projects (there were a lot more to select from). Tables 6.1 and 6.2 present the comparison of the tools.

The viewpoints of my analysis were:

- Database:

  The most important thing in the questions of the security is the efficiency of the tool. The bigger the database is, the more efficient it is. Here belong the environments which can be scanned with the tool, and the base (the place they get it) of the databases too.

- Installation and maintenance:

  These tools cannot be deployed as containers, and they have a lot of dependencies, so the installation can be difficult, and in some cases nearly unsolvable. The maintenance of the tool (so the installation of the updates, and fixes) can cause a lot of problems, if they are not done automatically.

- Support:

  In the world of security being up to date is necessary. If the project has no signs to update regularly, and they do not have proper support to help if a question comes up, then the tool cannot be used. I looked for documentation too, because the functioning can be understood based on it only.

- Scanning method:

  The other most important thing besides the database is the scanning method of the tool, they are in connection with each other: there are big differences between a known vulnerability and antivirus scanner, and a common best-practices checker. From the previous one expects protection, and from the latter one expects advices to keep the system safe.

- Communication with Docker:

  This viewpoint includes the communication with Docker for running containers, and the method of the static analysis of a container (the tool run a copy of the image in the background as a container, and analyse that).

- Resource demand:

  In some cases the manager nodes can be embedded devices too, so the resource demand of the tools can count. Especially if the tool can analyse running containers too (and this function will be used).

As can be seen, one of the tools (Notary) is not participating in the tables. That is because Notary is very different from the others: it is a tool for trusted image management. The base concept is that the developer can sign its content offline using his own keys, and then push it to a Notary Server, and with Notary Client with the proper key(e.g. sent through a secure channel) the user can download it. This solution can ensure the cryptographic integrity of an image.

In light of the use-cases I would recommend using Docker Bench for Security and Notary in every environment, because they realize functions what are necessary, and other tools cannot do, as well as they do not consume a lot of resources. But these two tools are not enough to keep the system safe. At least one static analyser is needed: for small and medium sized environments I think Dagda is the best solution, because it has the smallest resource demand,it is easy to deploy it, and it is possible to analyse running containers with it too (but I still do not suggest this). For huge sized systems I propose to use Anchore: the difference between Anchore and CoreOS Clair is that Anchor has the option to analyse running containers, and it is better documented than Clair. But of course, if one is completely sure about not using runtime analysis, then Clair may be the better option, because it has a bigger community, and more frequent updates than Anchore.

	Anchore Engine	CoreOS Clair	Dagda	Docker Bench for Security
Database	NVD(CVEs), Software package vulnerabilities, Distribution specific vulnerabilities(Alpine, CentOS, Debian, Oracle, Red Hat Enterprise, Ubuntu)	NVD(CVEs), Distribution specific vulnerabilities(Debian, Ubuntu, CentOS, Oracle, Amazon, Alpine, SUSE OVAL)	NVD(CVEs), BugTraq, The Exploit Database (Offensive Security), Red Hat Bug Advisor, Red Hat Security Advisor, Distribution specific vulnerabilities (Red Hat, CentOS, Fedora, Debian, Ubuntu, OpenSUSE, Alpine)	CIS Docker Benchmark
Installation and maintenance	Has collected requirements to easy deployment; does not have automatic updates	Complicated; working Go environment required; has automatic updates	Has collected requirements to easy deployment; does not have automatic updates	Can deployed as a container (so easy deployment, and auto updates)
Support	Well documented; frequent updates; active community; has 0-24 support	Big, active community; very frequent updates; not well documented	Formerly frequent updates, but nowadays looks abandoned (no update for half a year); small community; not well documented	Frequent updates; no documentation at all; small community
Scanning method	Known vulnerability scan for images, and runtime security analysis	Known vulnerability scan for images	Known vulnerability scan for images, and runtime security analysis(with Sysdig Falco)	Checks common best-practices around container deployment (agains images and containers too); Not a vulnerability scan!

Table 6.1: Part 1 of comparison of security tools

	Anchore Engine	CoreOS Clair	Dagda	Docker Bench for Security
Communication with Docker	Static analysis: simulate the running of the image and scan for vulnerabilities; Runtime analysis: communicates with Docker API	Static analysis: simulate the running of the image and scan for vulnerabilities	Static analysis: simulate the running of the image and scan for vulnerabilities; Runtime analysis: communicates with Docker API	Communicates with Docker API
Resource demand	Very high (at least 4 GB of RAM), high CPU usage during scanning	High CPU and RAM usage during scanning (more than 4 GB ram required)	Moderate CPU and RAM usage during scanning (at least 2 GB of RAM needed)	Almost nothing CPU and RAM usage

Table 6.2: Part 2 of comparison of security tools

Chapter 7

Conclusion

My goal with this thesis was to present that embedded devices can use, and has to use containerization. I kept in mind the price of the solution at every consideration, because the most attractive in an embedded device is the low price, and with an expensive solution this will terminate. My starting base was the problems of the nowadays used IoT devices. These include the question of security, and the management of the devices as a connected system.

The number of the IoT devices shows strong growth, thus these problems will become worse. The usage of virtualization is a possible way to solve these. But as I mentioned in Chapter 2, virtualization cannot work on embedded devices because of the lack of resources, however container based virtualization works in a different way, the resource demand of it is low, so it works well on embedded systems.

Docker is a de facto standard in the world of containerization, so I started to work with it. The first solvable question was the maintenance of the separate containers as a coherent system. The solution for this is the usage of an orchestrator. Not every orchestrator works well with these devices (because of the system requirements), so finally I decided to use Docker Swarm, because it is a free to use tool, and it is basically available on every device, which Docker Engine version is bigger than 1.12.

In Chapter 4, I looked for use-cases for the usage of Docker on embedded devices. Based on this chapter, I recommend using containerization on every embedded environment, because it makes the deployment faster, and can work well in every use-case. I gave advice for every scenario relative to the architecture of the system.

Therefrom the containers are handled as a coherent system, the management of the devices do not become easier. Docker has a lot of tools that can help with in, so I tested a few open-source management tools. There are essential differences between them, so I compared them in a table. Finally, I gave recommendations for every use-case.

Finally, I discussed security questions. The main part of the problem disappears with the usage of Docker. On the one hand, it is because Docker runs the containers in isolation, and what is more the containers and the images can be scanned with security tools. Because of the device's resource limits, only static analysis of the images can be a good way to defend the system. I tested tools with this functioning, and I compared them in a table form, and based on the results I recommended tools foreach use-cases.

This solution is not complete, because of the security issues around Docker. Only root, or a person who is part of the 'Docker group' (which is nearly equal with root) can run containers on a system. So if an attacker can break out from the container, then it will have root/nearly root privileges on the host system. The usage of SELinux can ensure limited access in such scenarios through policies. The system test with SELinux on embedded devices with an outbreak from a container can be done as future work, to complete all of the security questions around Docker/ around running containers on IoT devices.

Docker's checkpoint function can be used to restore the system to a stable state. But this is only an experimental feature, so this can be tested on embedded devices as future work too. As well as there are a lot more types of tools (e.g. monitoring, logging, networking), that are secondary importance, but still very useful, they all can be tested in the future.

Acknowledgements

First of all I would like to thank Dorottya Futóné Papp and dr. Levente Buttyán, leader of the CrySyS laboratory at Budapest University of Technology and Economics for the lot of help, and advice, with which they contributed to my work.

The presented work was carried out within the SETIT Project (2018-1.2.1-NKP-2018-00004)[1].

[1]Project no. 2018-1.2.1-NKP-2018-00004 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.

Bibliography

[1] Anchore. Anchore open source engine, Accessed December 5, 2019. URL https://anchore.com/opensource/.

[2] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, Vancouver, BC, August 2017. USENIX Association. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis.

[3] Thanh Bui. Analysis of docker security. *CoRR*, abs/1501.02967, 2015. URL http://arxiv.org/abs/1501.02967.

[4] T. Combe, A. Martin, and R. Di Pietro. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62, Sep. 2016. ISSN 2372-2568. DOI: 10.1109/MCC.2016.100.

[5] Dagda. Dagda github, Accessed December 5, 2019. URL https://github.com/eliasgranderubio/dagda.

[6] Docker. Docker healthcheck, Accessed December 5, 2019.. URL https://docs.docker.com/engine/reference/builder/#healthcheck.

[7] Docker. Docker website, Accessed December 5, 2019.. URL https://www.docker.com/.

[8] DockStation. Dockstation github, Accessed December 5, 2019. URL https://github.com/DockStation/dockstation.

[9] Black Duck. Black duck security, Accessed December 5, 2019. URL https://www.blackducksoftware.com/.

[10] Docker Bench for Security. Docker bench for security github, Accessed December 5, 2019. URL https://github.com/docker/docker-bench-security.

[11] HashiCorp. Hashicorp vault project, Accessed December 5, 2019. URL `https://www.hashicorp.com/products/vault/`.

[12] B. I. Ismail, E. Mostajeran Goortani, M. B. Ab Karim, W. Ming Tat, S. Setapa, J. Y. Luke, and O. Hong Hoe. Evaluation of docker as edge computing platform. In *2015 IEEE Conference on Open Systems (ICOS)*, pages 130–135, Aug 2015. DOI: `10.1109/ICOS.2015.7377291`.

[13] G. Kambourakis, C. Kolias, and A. Stavrou. The mirai botnet and the iot zombie armies. In *MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)*, pages 267–272, Oct 2017. DOI: `10.1109/MILCOM.2017.8170867`.

[14] *Common attacks on IoT devices*, October 23 2018. The Linux Foundation, Christina Quast. https://elinux.org/images/f/f8/Common-Attacks-on-IoT-Devices-Christina-Quast.pdf.

[15] Russ McKendrick and Scott Gallagher. *Mastering Docker*, chapter Portainer, pages 199–226. Packt Publishing Ltd, 2017.

[16] Russ McKendrick and Scott Gallagher. *Mastering Docker*, chapter Docker Swarm, pages 173–198. Packt Publishing Ltd, 2017.

[17] Notary. Notary github, Accessed December 5, 2019. URL `https://github.com/theupdateframework/notary`.

[18] Core OS. Core os clair documentation, Accessed December 5, 2019. URL `https://coreos.com/clair/docs/latest/`.

[19] Portainer. Portainer community edition, Accessed December 5, 2019. URL `https://www.portainer.io/products-services/portainer-community-edition/`.

[20] Mathijs Jeroen Scheepers. Virtualization and containerization of application infrastructure: A comparison. In *21st Twente Student Conference on IT*, volume 1, pages 1–7, 2014.

[21] Aqua Security. Aqua microscanner github, Accessed December 5, 2019. URL `https://github.com/aquasecurity/microscanner`.

[22] Randall Smith. *Docker Orchestration.* Packt Publishing Ltd, 2017.

[23] Swarmpit. Swarmpit github, Accessed December 5, 2019. URL `https://github.com/swarmpit/swarmpit`.

[24] James Turnbull. *The Docker Book: Containerization is the new virtualization*, chapter Introduction, pages 6–7. James Turnbull, 2014.

[25] Docker Compose UI. Docker compose ui github, Accessed December 5, 2019. URL `https://github.com/francescou/docker-compose-ui`.

[26] Yuyu Zhou, Balaji Subramaniam, Kate Keahey, and John Lange. Comparison of virtualization and containerization techniques for high performance computing. In *Proceedings of the 2015 ACM/IEEE conference on Supercomputing*, 2015.