Enhancing and integration of security testing in the development of a microservices environment

Master of Science in Technology Thesis
University of Turku
Department of Future Technologies
Security of Networked Systems
September 2020
Gabriele Orazi

Supervisors:
Karo Vallittu (Awake.AI)

Examiners:
Petri Sainio (University of Turku)
Seppo Virtanen (University of Turku)

UNIVERSITY OF TURKU
Department of Future Technologies

GABRIELE ORAZI: Enhancing and integration of security testing in the development of a microservices environment

Master of Science in Technology Thesis, 85 p.
Security of Networked Systems
September 2020

In the last decade, web application development is moving toward the adoption of Service-Oriented Architecture (SOA). Accordingly to this trend, Software as a Service (SaaS) and Serverless providers are embracing DevOps with the latest tools to facilitate the creation, maintenance and scalability of microservices system configuration.

Even if within this trend, security is still an open point that is too often underestimated. Many companies are still thinking about security as a set of controls that have to be checked before the software is used in production. In reality, security needs to be taken into account all along the entire Software Development Lifecycle (SDL).

In this thesis, state of the art security recommendations for microservice architecture are reviewed, and useful improvements are given. The main target is for secure to become integrated better into a company workflow, increasing security awareness and simplifying the integration of security measures throughout the SDL.

With this background, best practices and recommendations are compared with what companies are currently doing in securing their service-oriented infrastructures. The assumption that there still is much ground to cover security-wise still standing. Lastly, a small case study is presented and used as proof of how small and dynamic startups can be the front runners of high cybersecurity standards. The results of the analysis show that it is easier to integrate up-to-date security measures in a small company.

Keywords: microservices, containers, security, testing, software development lifecycle, SaaS, Serverless, Service-Oriented Architecture

Contents

1 Introduction

Nowadays, each of us benefits from a variety of services available online by accessing them through different terminals. By registering for the service it is usually possible to access the same functionalities, as well as the same data. This has allowed during the last decade to decrease the need of using the same device in favor of a greater dependence on the service and its related data. Also the cyber security world has been affected by the change, shifting more and more the attention towards web applications and the data flow generated by them. As stated by the 2020 Verizon Security Report[1], nowadays more than the 90% of breaches have web applications as hacking vectors, confirming the growing trend if compared to the same report from previous year[2]. Indeed, this information confirms that many of the applications that few years ago were installed on a desktop, nowadays can be used from a standard web browser.

During the decades, the growing demand of software challenged software architects to came up with designs able to meet requirements and, at the same time, be resilient enough to support changes. Thus, code bases irremediably experienced an exponential growth in size, complicating more and more the management of the product itself. This created the need for a new solution that would allow the complexity of the product to be separated, making it easier to develop and manage. Microservice architecture was the answer to this need, the one which has been able to solve many problems that businesses were struggling with. Despite the benefits

that this technology brings to organizations today, the implementation of this architecture can be more complex. In parallel, the safety of the platforms that take advantage of this design has also had to adapt to the new introduced challenges.

This study seeks to address the management of safety in the new ecosystem resulting from the combination of the two trends described above. The Web application based on microservice architecture is today the most advanced and widespread solution in software engineering. Ensuring this technology as accurately as possible is therefore a subject that deserves to be explored in depth and is, moreover, the aim of this study. Thus, the thesis will try to understand which are the best specific tools that have been created or adopted specifically for microservice environments. The market demands for secure platforms and companies should adopt (or aim for adopting) the most up-to-date recommendations to satisfy such requirement. For this reason, framing how companies are doing security-wise speaking will be another objective of this thesis. Nevertheless, an investigation will be conducted into how the size of the business can influence the cyber security of the web applications that the companies themselves offer on the market.

The second chapter of the thesis provides a wide understanding over the methods, tools and techniques that will often be recalled in the paper. In the third one, a complete and up-to-date overview of the elements that allows companies to increase security among all the software development process is offered, while in the fourth chapter some innovative methodologies are proposed as integration of the previously explained framework. The fifth chapter's aim is to compare the provided set of best practices with what is currently implemented and used by average in tech companies panorama. Lastly, a case study of a small startup and its security vision is analyzed to understand strengths and weaknesses that derives from its small structure. Conclusions close the thesis, using the results of the analysis to answer the open points raised initially and proposing future developments for this work.

2 Key principles and tools for Agile Fast Development Workflow

In order to describe and study Information Security (InfoSec) and its application in the microservices environment, it is important to clarify the context and some main principles that will be useful to take for granted in the development of this work. Even if the next sections may not be directly connected with InfoSec itself, the definition of these assumptions are mandatory for the development of this study.

In the next sections *security engineering* aspects will be taken into account, as well as some related Software Development Operation (DevOps) best practices and basics. Main tools used in *Microservices* environment will be also described.

2.1 Agile security engineering method

The current security engineering found its baseline in the Agile software development. Starting from 1990, companies started to shift from a traditional and inefficient software development to a more sophisticated and comprehensive method. As examined by researchers Hoda, Salleh and Grundy[3], Agile introduced practicalities and processes to boost the level of coordination within the teams, increasing productivity and, more importantly, adapting the developed product based on the stakeholder's willing. The Agile Alliance, guided by Kent Beck, manage to enclose such a big and powerful development methodology into few solid principles collected

in the famous Agile Manifesto[4]. Each of them represents an essential milestone where all Agile-based approaches need to base their philosophy.

According to the 13th Annual State of Agile Survey[5], in 2018 the 97% of all the interviewed organizations reported to adopt agile methodology at least in one team within the company, demonstrating the growth of the trend if compared with the 84% of the first version of the survey, made in 2007[6].

During the years, the Agile concept has been interpreted in many ways, giving rise to different specific methods. Despite the proliferation of proposals, few of them have survived in the past decades. In particular, the Annual State of Agile Survey[5] states that Scrum-based agile methodologies are still the most widely adopted with a 72% of the 1319 people interviewed coming from all different kinds of organizations. Few other methods populate the rest of the ranking, but the most relevant is *Kanban*[7]: it is adopted as *Scrumban* (combination of Kanban and Scrum with 8%) and *Kanban* itself (5%).

The co-creator of SCRUM, Jeff Sutherland, explains in his book[8] how such process can be effective in plenty of different contexts that can easily lie outside from software development. Also the previous mentioned paper[3] underline the spreading of the Agile style in disciplines beyond software: project management, marketing, human resources or sales are just few good examples that sustain the statement.

Security engineering represents a field in which Agile methods can be applied with enormous gains, especially if in conjunction with the software development flow. An interesting study[9] made by K. Rindell, S. Hyrynsalmi, and V. Leppänen took into account many different agile methodologies in order to analyze how security perspective has been evolved and adopted within the methods itself. Among the plethora of processes, the authors pointed out the slowness of the development of such methods in security-wise perspective. Despite such delay, methodologies

nowadays include security as important milestone of the process. Perhaps, the study sadly highlights the detachment of theoretical usage from actual empirical implementation: all the case studies that have been reviewed by the paper[9] shown that reality is far from been compliant with security development recommendations. In 2017, another research[10] has been conducted with the aim of sketching how security engineering is performed within different kind of Software Development Lifecycle (SDLC) (more about the topic in the next section 2.1.1). For the purposes of this paper, the most relevant extracted insight is that the 42% of the 118 analyzed primary studies concentrate security controls in *code phase*, relying merely on static and dynamic analysis on top of the written code. Once again, this proves the grade of inefficiency regarding the real implementation of security engineering.

In the next sections, a description of the fundamental principles for agile security engineering are explained. The aim of this chapter is create a solid background for the following chapters, that will have to refer to the notions explained here.

2.1.1 The Software Development Lifecycle in Agile

As well resumed by Nayan B. Ruparelia in his paper[11], different models can be adopted as guidance for developing code within a team. New approaches have been proposed during the years, many others have been rediscussed and improved. The so called Software Development Lifecycle (SDLC) usually defines different stages of the development and how to go through them. The first documented SDLC has been the *Waterfall Model*[12]: even if it has been widely adopted since its publication, the model nowadays is almost neglected. The Waterfall method is too rigid and doesn't provide the possibility to make changes in the design. Development steps are followed in a row, completely omitting on-going discussions and adaptations of the project together with the stakeholders. Regarding Waterfall approach, Ericsson AB in Sweden was taken as test case in 2009 for studying the methodology. The

resulting paper[13] evidenced all weaknesses and reasons why such method cannot be considered as effective in software development anymore.

Eventually, many other methods have been released after this one: *B-model*, *Incremental model* and *V-model* are some good examples[11]. The central idea introduced with those is the concept of cycling through stages, leveraging the importance of dynamic definition of the goals in the short term. This approach has been the key that enables organizations to solve problems of the previously adopted Waterfall-like methods. Development is split in many sprints that usually last no more than 3-4 weeks. Even if they may slightly vary, all these cycle-based approaches follow **five basic phases** within each sprint:

- **Requirements**: user stories are generated and translated into features to be implemented;
- **Design/Plan**: upcoming features to be developed are selected and discussed; goals are split in small tasks and prioritized;
- **Develop/Code**: tasks are accomplished;
- **Test**: fresh new features are stressed in order to detect possible problems/flaws;
- **Deploy**: developed features are tested and integrated into the product.

In fact, all the *Agile techniques* mentioned in section 2.1 are compliant with this philosophy.

2.1.2 The Shift Left paradigm

Differences between teams always leads to interesting insights if seen from the correct perspective: in the Accelerate annual State of DevOps 2019 report[14], many different teams has been taken into account thanks to more than 1000 respondents.

The report divides teams into four main groups, from *elite performers* towards the *low* ones. Thus, the comparison between elite and low performers shows that the best ones have 7 times lower change failure rate. What is really relevant from this information is the adopted process before a change: elite groups are not necessary less error prone than the low ones, but have the capability to capture possible point of failure before than the others, avoiding to silently carry the errors until the deployment. This is the main key ability that stands behind the *shift left paradigm*: addressing issues (including security related ones) towards the early stages of the SDLC (discussed in section 2.1.1).

In support of the above, the *Error cost escalation analysis through the project life cycle*, by NASA[15] in 2004. Taking into consideration three different methods of analysis and comparing them with previous accomplished studies in the subject, the study clearly demonstrates that the cost escalation has an exponential growing trend. The representative outcome of the paper can be identified in the multiplicative factors shown in table 2.1. Software and electronics hardware (system) projects have been compared, showing that despite the apparent misalignment of results, the order of magnitude of growth does not change.

Therefore there is no doubt that waiting for validation coming from the *testing phase* is already too late, generating a greater required effort in solving the issue. Even if performing security controls sounds like something to be performed in the late steps of the development, ensuring information security can gain from the shift left principle as well. Each stage of the lifecycle requires different approach (as discussed in chapter 3), which needs to go hand in hand with usual software development process.

	Software Cost Factors	**Systems Cost Factors**
Requirements	1X	1X
Design	5-7X	3-8X
Build	10-26X	7-16X
Test	50-177X	21-78X
Operations	100-1000X	29-1615X

Table 2.1: Comparison of software & systems error cost factors by NASA study[15].

2.2 Serverless

Even if the usage of *Serverless* configurations is rooted in 1995, the diffusion and improvement has grown constantly among decades until today. In the first complete publication[16] about the topic, released by University of Berkeley, the authors managed to craft the first working configuration of a serverless solution combining different technologies such as RAID, LFS and Zebra.

During the years, system architectures has grown, pushing further the potential of the technology itself and, at the same time, overcoming bottlenecks of centralized file system design. Serverless is just the last stage of a process which has started with old non-virtualized servers, passing through the event of *hypervisor*[17] for virtualization and *docker*[1] for containers. The general tendency of shifting to Platform as a Service (PaaS) concept influx on the success of Serverless adoption, which nowadays is able to offer scalability, high availability and the easiest management ever reached until now.

Eventually, the most advanced stage of server configuration has been reached during the last years with the cooperation of container technology combined with Serverless architecture, giving rise to the container-based microservice system, dis-

[1] `https://www.docker.com/`

cussed more in detail in chapter 2.3.

2.2.1 Advantages

The success of the technology relies on few but fundamental points of strength that allow many *"agile"* realities (from the smaller start-up to the biggest enterprise) to enhance the effectiveness of fast pace development discussed in section 2.1.

The most tangible effect for the company that uses the service is the complete **disappearance of low-level server management**. The provider of the serverless solution takes charge in all the responsibility, usually offering an User Interface (UI) where to customize and setup the required configuration. In addition, monitoring tools for the service are completely focused on information regarding the service itself rather than the hardware that actually is running it and its architecture. Access control alerts, error messages and traffic analysis are good instances of what would be shown as metrics, taking the place of CPU and memory consumption.

Withstanding service requests is the basis of reliability, therefore the ability to **scale seamlessly and automatically** is one of the main reasons of the wide adoption of serverless solutions. The cost and the difficulty of handling manually a reasonable level of scalability is already enough to justify the adoption. Service providers are able to adjust dedicated resources with respect to the load of the service, relieving engineers of the onerous task of managing resources.

Providers of the service are always able to offer to their customers an **high availability by design**: dynamic allocation of resources requires relying on a complex distributed system of redundant machines that is able to continue operating also in deteriorated situations.

In business terms, companies who decided to adopt a serverless solution may end up in saving costs, since the service is supplied with a **bill-per-request model**. It is fair to point out that this favourable condition may not always be met as it is

dependent on the load of requests. Small and medium companies may gain the most from these agreement, big enterprises may prefer to invest resources to handle their own infrastructure internally.

2.2.2 Disadvantages

Even if advantages are usually in favour of every possible scenario that can take into consideration such architecture, some drawbacks can be identified as well.

The **vendor lock-in** phenomenon is the biggest concern for companies who sign contracts with service providers. Once the service has been configured it may be really difficult to migrate into another provider because of technical and businesses aspects. The shift would require a massive adaption that could be evaluated to be more expensive than the gains generated by the shift itself. Moreover, each service provider offers their own small services that are well integrated with one another, discouraging the adoption of third party services and implicitly pushing the customer to rely on a single master vendor.

Related to this, customers will necessarily end up in **losing control over the environment** and the underlying infrastructure. Exposing the customer to disasters which may happen to the vendor such as exposed vulnerabilities resulting in security breaches or power outages generated from natural disasters.

2.3 Microservices

A microservice can be defined as a small application that can be tested and deployed independently, as well as scale to satisfy the demand load. The most important principle to keep in mind is the **single responsibility** one, which imposes that each microservice must provide a simple, defined and atomic service. As stated by Johannes Thönes in his interview published by IEEE Computer Society[18], mi-

croservices approach goes against the old-fashion **monolithic systems**: rather then include all features of a product into a massive single piece of code, microservices are aiming for splitting applications into smaller pieces. Each single fragment of the application is separately maintained but, at the same time, is able to cooperate together with the other components and, when not needed anymore, it can be easily shut down. On the same line as Thönes' words, also the Newman's book "Building Microservices"[19] states microservices method allows to make fast changes, which translate into faster deployments as well. Continuous Integration and Deployment (CI/CD) (described in section 2.4.2) represents a natural solution to enhance the potential of microservices, resulting in a more effective software delivery into production. In monolithic application, small changes need to be released together with the whole application. These kind of deployments are risky, therefore they cannot be performed every now and then. Increasing time in between one release and another increase the possibilities of incurring in errors. On the other hand, microservices permit handling possible errors without risking too much, thanks to the isolation between different services and the simplicity of rolling back to the previous version if needed.

It is difficult to efficiently scale a monolithic application since it has to be seen as a single piece that requires a certain amount of resources. Splitting services allows the usage of less powerful hardware and, even more importantly, to instantiate the amount of resources on demand. Either organizational alignment can benefit of the same idea: develop and maintain many small products reflects also in having smaller and more productive teams.

Even if it can be understood as simple, defining granularity of the services is not a trivial task: usually each service should stay within 2000 lines of code, but 200 lines could be more than enough. Since this definition may depend on too many factors, a better definition has been provided by Jon Eaves[19], which uses time

required to write from scratch the service as a metric, estimating two weeks as a correct period in which a service should take to be completely reinvented.

Since each service is decoupled one from another, it is possible to maintain a high level of **technology heterogeneity**: in other words, tools and languages can be decided depending on the specific needs of each service, helping in maintaining a lightweight stack and increasing performance. Moreover, this encourages the adoption of new technologies since those can be tested without big changes in the overall architecture. In a monolithic system, the substitution of each single component would require excessive effort and an high risk task to be performed, therefore discouraging developers to upgrade the system in favour of newer and better components.

The complete picture of microservices is obtained thanks to the creation of an effective **network communication**. Each single piece should be autonomous, exposing itself to the other nodes through simple Application Programming Interfaces (APIs). Citing Newman's book[19], an effective and simple question to point out in order to understand if a service is autonomous would be:

> "Can you make a change to a service and deploy it by itself without changing anything else?"

Usually, if the answer is *yes* a good design has been performed.

As all innovative technologies, microservices introduced challenges together with improvements discussed since now. Even if it is true that as the service granularity gets smaller the benefits increase, it is also reasonable that the amount of connected pieces increase the complexity. Spreading responsibilities into smaller components increase the difficulty of enforcing security measures against the overall application. Because of the heterogeneity of the services, an administrator will have to deal with different security domains, depending on tools and environments in use in each node. This exposes a greater surface of attack, requiring particular attention in the

definition of each single exposure point provided by APIs. It is also important to notice that with such an architecture, introducing the concept of trust is needed: if one component becomes infected and/or controlled by a malicious party, it can easily turn into a vector useful for compromising the entire application.

Concluding with another quote from the above-mentioned book[19]:

> "As you get better at handling the complexity, you can strive for smaller and smaller services."

In the next paragraphs, containers and their orchestration will be described. Those elements can perfectly represent the practical and tangible representation of the concepts delivered by microservices design.

2.3.1 Containers - *Docker*

Despite their novelty, containers have been considered from the very beginning as a powerful solution to solve a lot of technological challenges. Released in March 2013, **Docker** project is *de facto* implementation of containers. Thus, from now on, containers will be considered as Docker images and vice versa. The potential of Docker has been highlighted already in 2014, when Dirk Merkel wrote about the topic in the Linux Journal[20]. A lot of limits imposed by the usage of virtual machines have been pushed far away thanks to this technology: cloud computing is one of these fields, where the advantages have been promptly emphasized in the study made by Di Liu and Libin Zhao[21] already in the late 2014.

The official documentation[22] provided by the company itself is accurate and useful to understand the architectural implementation. Docker provides OS-level virtualization and is running as a system daemon that acts as an interface between the instantiated containers and the client that sends commands to the processes. The aim of a container is to sandbox a process. The container itself is a single process nested in the host machine kernel. The container itself can be considered as alive

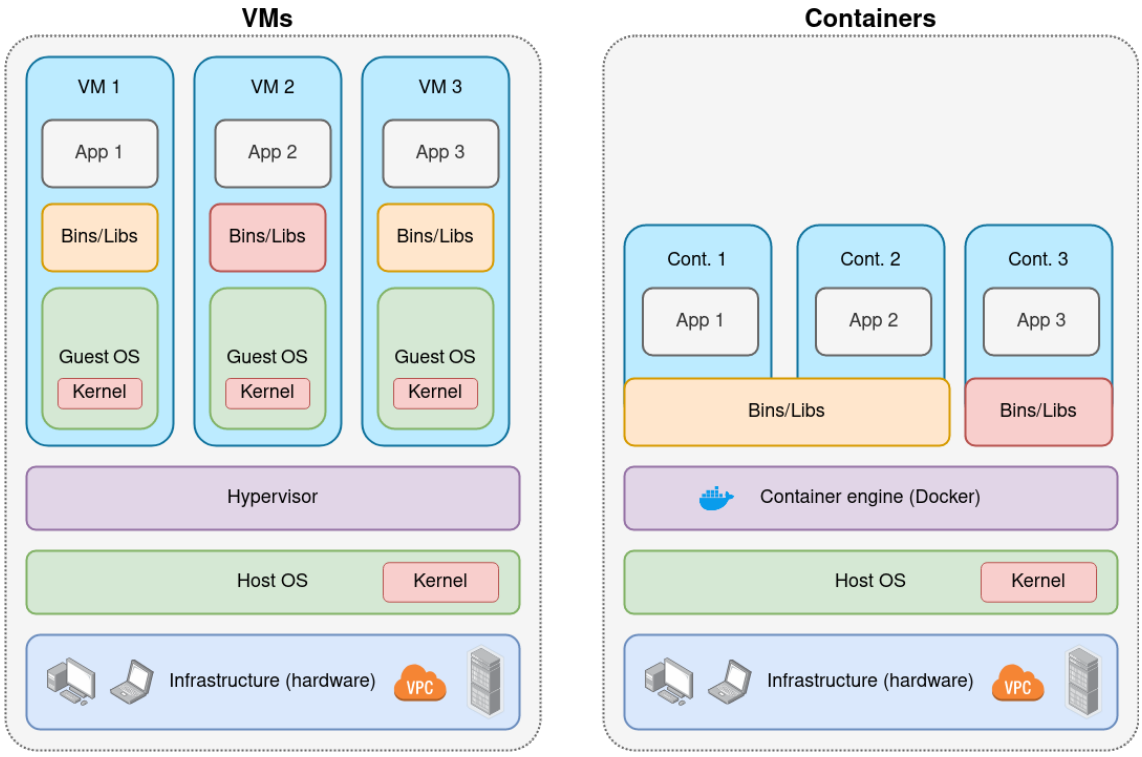

Figure 2.1: Architectural difference between VMs and containers[21].

as long as the relative process is running. The difference between a container and a virtual machine (VM) is as simple as effective: each single VM is a virtualization of a guest Operating System (OS), with its own kernel running; on the other hand, the strength of containers is the usage of the host machine kernel. The result is a single process which shares system libraries with the host machine but that at the same time is enclosed into a single process. Differences in the architecture can be seen in figure 2.1.

A **container image** is the binary representation of the container itself and is nothing more than a composition of different other images, organized in a hierarchical manner. Each image is a standalone and can be modified, updated and changed. If one image of the tree has been modified, all the relative descendants of that node can be rebuilt in order to adopt the latest changes.

Building a container image can be accomplished in two different ways:

- The *Dockerfile* is a `.json` or `.yaml` file which describes the image itself. The `FROM` field defines the parent image where to start and the rest of the file describes additions made to the parent.

- A docker image can be also build **from runtime**: it is possible to run the parent image, make changes and configurations to it and then generate a replicable image out of that specific execution.

Usually, Docker images are collected into registries. For instance, the *Docker Hub*[2] is the public one provided by Docker but it is possible to create also private registries where to store private images. This can be the use case for companies which don't want to share their own developed containers. The **Docker Host** is considered as the machine in which the daemon and containers are running. Images can be pulled and pushed from the registry and stored into the *docker host* cache registry.

The client manages the life cycle of the containers, interacting with the daemon. The client can:

- Push and pull images from the cache;

- Create new images;

- Build an image;

- Run an image;

- Configure the infrastructure of the docker host in which containers are operating, which means network and storage configurations.

Figure 2.2 helps the reader to visualize how the entities described before are related between each other.

[2] `https://hub.docker.com/`

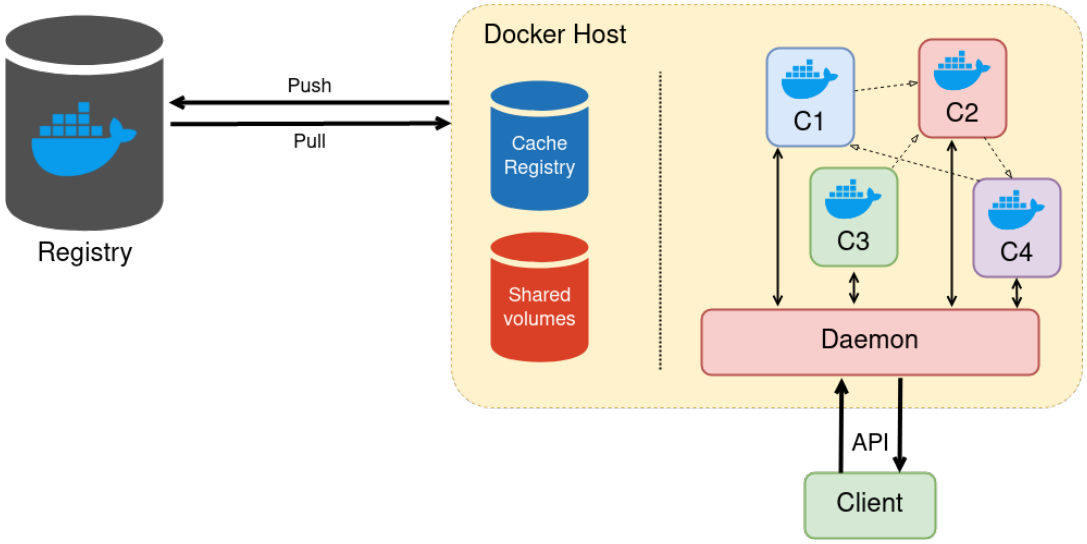

Figure 2.2: Docker host overall architecture[22].

There are no dependencies between the container and the host machine other than the daemon: everything is packed into the container itself: because of this, once a valid daemon can be installed in a machine, every container can run on top of it, independently from the host machine OS. This independence is based on two principles:

- Process **namespaces** provide isolation between different processes, limiting processes that a container can see, therefore limiting also the usage of those;
- **cgroups** (control groups) aim is to discipline how much a container can use a specific resource. Metering and limiting resources such as memory, CPU and network is the aim, providing also access control over devices.

2.3.2 Container orchestration - *Kubernetes*

Container orchestration is the next step that enables the concept of containers to enhance their potential. Even if containers can communicate between each other without being included in a cluster, these are not specifically designed to accomplish this task. It is important to notice that container clustering solutions can address

the majority of the security-related problems that have been discovered regarding possible attacks. In support of this statement, Docker has been analyzed[23] in 2016 in order to find possible conceptual security weaknesses: the authors of the research stated that container orchestration tools can solve all the discovered threats.

Google started a project written in Go, called Kubernetes (often shorten as K8s), which than has been donated in open source to Cloud Native Computing Foundation (CNCF). Similarly to Docker, Kubernetes quickly assumed the role of the standard tool used for the purpose. Kubernetes technology is almost in all companies and organizations who decided to use microservices: in 2018, *TheNewStack.io*[3] conducted an investigation[24] discovering that Kubernetes was adopted in the 69% of organizations with microservices architecture. Moreover, the following positions in the ranking were occupied by Kubernetes wrappers such as Amazon ECS or Google Container Engine. For this reason, as for the containers, this study will refer to Kubernetes as the container orchestrator and viceversa.

The term "orchestration" stands for the ability to organize and coordinate efficiently multiple small services provided via container technology. As explained in the official online documentation[25] and shown in figure 2.3, a cluster contains basically three actors:

- The **Master** (usually single, but for complex configurations may also be more then one) is the element which coordinates the work in between nodes and it should be free of normal workload and completely dedicated to receive requests, schedule tasks to assign to nodes and handle shared resources;
- **Nodes** handle the workload and execute tasks assigned by the *master*;
- A **Pod** is the atomic unit of Kubernetes. It is always contained inside a *node*. Pods include container(s) and provide to those a shared sandboxed

[3] `https://thenewstack.io`

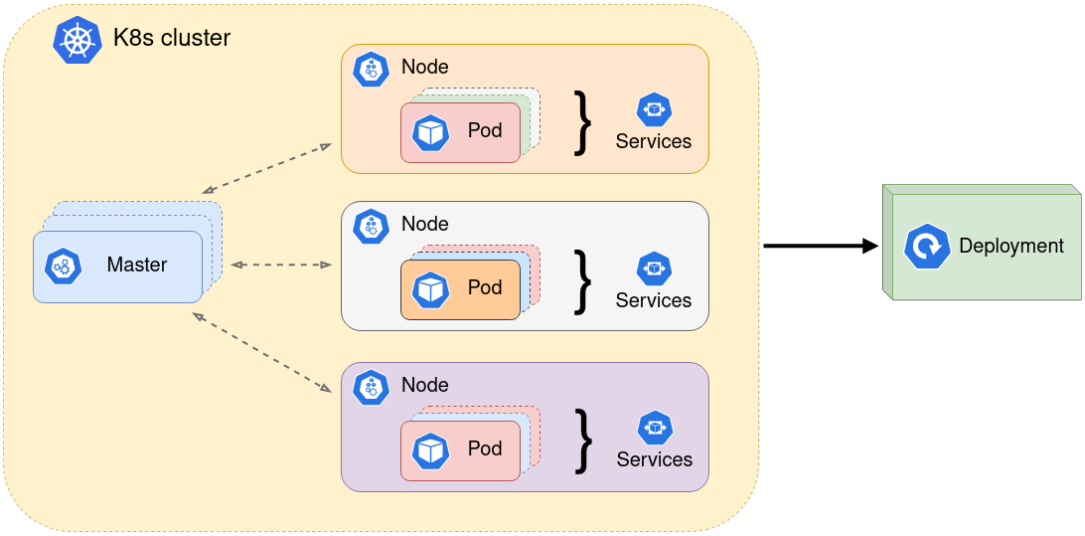

Figure 2.3: Kubernetes architecture[25].

environment in which to operate. Pods do not have a state and they follow a simple lifecycle where they born, live and then die when they have served their purpose; they can be easily replicated and included into a **ReplicaSet**, which is a collection of different instances of the same pod;

- A **service** is the element that provide a unique IP/DNS for a group of pods that are working together. Services and pods are coupled thanks to a simple and effective *labelling* system: pods are described with a set of labels (version, purpose, environment, ...). A service is defined through a set of labels as well, which are used to claim the belonging of such pods who match its categorization;

- A Kubernetes **deployment** has to be intended as a self-documented declaration of the cluster structure which is intended to be created and maintained during runtime; it is defined with a *manifest* file (either `.json` or `.yaml`); the most important element specified into this file is the number of *ReplicaSets* to be maintained as active during the execution.

The capabilities of this architecture permits Kubernetes to be considered as one

of the optimal solutions in terms of *scalability*.

2.3.3 DIE paradigm

The simplicity of a pod lifecycle is in line with the DIE paradigm. The short lifespan allows the orchestrator to handle more easily its atomic units, spawning or killing instances depending on the workload. Furthermore, as soon as the pod reach an error state or, for some reason, it gets stack, Kubernetes is able to detect the faulty state and proceed with killing the single unit.

Each pod is coupled to the specific release of the software with which it was created and there is no way to update or change it. This concept of *immutable software* wants to enforce the idea that none of the pods should have a long fixed life. Whatever is the patch, update or error that has to be handle, the solution would always be to kill and create a brand new instance with changes up from the very beginning. Being in line with this idea should push developers and DevOps to bear in mind that the service should be as lean and clean as possible, with specific and well circumscribed responsibility.

2.3.4 Distroless containers

By their design, containers should provide a lightweight solution to run code. As already discussed, each container is based on a base image specified in the `FROM` section of the `Dockerfile`. Although slim Linux distribution (such as *Alpine Linux*[4] or *Debian*[5]) are used as baseline for deploying a service, still a lot of unused and unnecessary dependencies are introduced. Those dependencies usually are deployed together with the application that the container is supposed to provide, inevitably increasing the size of the image and, even more importantly, opening the side to

[4] `https://alpinelinux.org/`

[5] `https://hub.docker.com/_/debian`

possible attack vectors.

Google aimed to address such problem introducing a new concept of **Distroless docker images**[26]. The pioneering project aims to reduce the content of the final deployed image to the application itself, the associated resources and the minimum amount of language-related dependencies needed at runtime. Thus, the eligible attack surface exposed will be reduced, as well as the size of the image. As a side-effect, also the overhead needed to scan and patching vulnerabilities related to unused dependencies will be cut down.

In addition, this compression offers the chance to developers to try to provide a *read-only* image whenever possible; getting rid of *shell* access is also feasible, reducing again the attack surface. Even without a shell, debugging the image will be possible through the usage of *multi-stage builds*[27]: *Docker* has introduced such feature in order to help the optimization of images. Before this feature, developers who were aiming to deploy small images usually had to work with a big container with all kind of needed dependencies and tools for developing. Once the image was ready for deployment, a new tiny image was crafted, trying to reduce the content to a minimum. With the introduction of *multi-stage builds*, developers are able to avoid the use of two different images simply using multiple `FROM` statements in the image manifest file and giving the possibility to pick one version at build time.

In a big picture such as a Kubernetes cluster, where a lot of pods need to be instantiated and where connection between them may represent the biggest security concern, *distroless containers* represents one of the best and most effective mitigations against possible attacks.

2.4 Release Management

The integration of brand new features and patches into the final product is a vital topic that each organization needs to take into account. Despite each company may

decide to adapt the releasing process with respect to their specific needs, keeping trace and versioning a product is an essential activity which has be agreed within the entire team.

Release management follows the entire development of a software build through its planning, scheduling and controlling phases, until hitting the release. Usually software is deployed across different environments in order to be tested and then, consequently, pushed into production. In a context in which Agile methodologies (discussed in section 2.1) are embraced, techniques for establishing some level of automation whenever a new feature is ready to be tested has been created.

These will be the topics described in this section, always keeping attention on a *security perspective* and its real applicability within the deployment process.

2.4.1 Deployment environments

Whenever a product needs to be delivered and maintained to the customer, an organization needs to mitigate the risk of producing unexpected disruptions of the provided service. Developing new features or patching problems is a high risk operation to be directly performed on the customer product. For this reason it is important to add some layers in between the developer machine, where the features are tested in the first place, and the machine which actually serves the customer's product. Therefore, creating a separation in between a *development* environment (*DEV*) and *production* one (*PROD*) is necessary. This separation allows developers to deliver and observe new features in a not exposed replica of the product and then, release or roll back the code depending on the outcome of the tests. The usage of many other intermediate steps in between DEV and PROD can be adopted in order to increase the pace of released features together and reducing at the same time the risk of production failure.

Even though the configuration of such an infrastructure may appear to be com-

plex to configure and maintain, nowadays Serverless solutions (described in 2.2) integrate such functionalities into their products. Thanks to this level of simplification, it is quite easy for DevOps to handle permission, resources and fine-tuned configurations through a simple control panel, leaving aside practical hardware issues and focusing on the logic.

There is a quite large grade of variation in nomenclature that each organization uses to apply to different environments in use. For this reason, in the following description of the main environments that usually are used within organization, the reader is invited to focus on the purpose of each environment rather than on the assigned name.

- **Development environment (DEV):** Usually developers use to work locally in their machine while implementing a new feature or a patch. As soon as the code seems to be mature enough, the next step is integrating this into the *development environment*. This environment contains all the newer implementation of the product but is highly failure-prone due to immaturity of the code. For this reason, disruption occurs often in DEV, but that doesn't represent a problem since this is the real purpose of this environment: provide an early test bench were to push changes without concerning too much about possible side effects.

- **Stage environment (STAGE):** This layer is often renamed as *Integration* environment. This version of the product is the collection of all the new features that have been tested singularly in DEV and that have been considered to be mature enough to be merged into the master branch. Therefore, in STAGE all the services and features are labeled as the latest version. The outcome of this environment may be still unstable due to the integration of all the new different features.

- **Quality Assurance environment (QA):** Whenever the organization decides to release a new update of the product, the current status obtained in the STAGE is assigned to a *tag* that express the *version* of the release. This specific frame of the product is supposed to be ready for being shipped into production. Thus, organizations may want to introduce an additional step in order to run more tests. This is the purpose of the QA, where a *release candidate* is produced and submitted to the testing team in order to assess that all the functionalities included in the release are guaranteed.

- **Production environment (PROD):** The product which is directly in use by the customers is the one contained in PROD. Since this is the real-life exposed version of the product, robustness is required and therefore releases represent a really sensitive step to be accomplished. Deploying errors or bugs exposes the organization to huge risks in terms of credibility and finance. When major releases are deployed, traffic routing may be required in order to switch from a version to the next one in order to avoid service downtime. Once again, Serverless solutions simplify such process, often offering automatic shifting processes. Big companies with enormous amount of users choose to restrict the release of the new version only to a small fraction of users in order to mitigate possible errors in the delivered version or even to collect feedback from users. As described in the "Continuous Delivery" book by J.Humble and D. Farley[28], this technique - called *canary releasing* - leads to good benefits: rolling back to the previous version would be much easier and observing how and how much the new developed features are used can also give a direction for future developments. Moreover, this method is considered to be really effective for testing capacity requirements, releasing the new version gradually to users.

The pace of the deployments, whether for STAGE, QA or PROD environments,

is related to the organization decisions. For internal stages, decisions are instructed by the team organization and the adopted development method, whether for public releases (and release candidates) it may be effective to take into consideration the nature of the product itself and the kind of user needs and priorities.

2.4.2 Continuous integration & continuous deployment

In the general Agile and fast development panorama (discussed in section 2.1), CI/CD plays an important role. The combination of these two processes enables a company to deliver a more complete and safer product with faster pace. Moreover, it releases developers from manual and inefficient tasks. CI/CD is supposed to follow the evolution of the code all along the different environments (considered in section 2.4.1), speeding up the entire process and, therefore, hitting the production deploy as soon as possible. If in the Accelerate 2019 Report[14] the conjunction of Development and Operation has been evaluated to be the game changer in software development, CI/CD represents the higher point of contact between these two worlds.

Already in 2017, the Systematic Literature Review conducted by M. Shahin[29] was highlighting the increase in interest in the topic. The paper took into account 69 papers published from 2004 to 2016, and the 56,5% of those were published in the previous three years. The growing trend has been discussed in section 5.4, confirming the importance of this practices. Shahin[29] identified seven main factors that impact the success of CI/CD, which are:

- *Testing (effort and time)*
- *Team awareness and transparency*
- *Good design principles*
- *Customer*

- *Highly skilled and motivated team*
- *Application domain*
- *Appropriate infrastructure*

Such factors will be emphasized in the next paragraphs, where a definition of Continuous Integration and Deployment is provided. It is important to define boundaries of these two methods that always work in symbiosis, but which too often are erroneously considered as the same thing. In support of these, some space will be dedicated to Pipelines, as the most used and effective tool.

Continuous Integration (CI)

In the traditional old fashion workflow, developers use to work on their own features for a long time. If everything goes smoothly, they start to think about integrating their feature after one or two weeks, even though there is the possibility that the isolation period can be also extended for a month, depending on the entity of the developed feature. When the feature is completed, the developer is called to merge the brand new feature into the master branch of the project. Unfortunately, going into this process will be so stressful due to the multitude of conflicts that the developer will have to solve before accomplishing the task. Moreover, the developer will also have to involve other colleagues to resolve the "merge hell", since he will have to deal with someone else's work, avoiding to break what as been done so far. As it might be evident, such process is time-consuming, stressful and expensive, as well as error prone. Developers need to probably deal with old code with a logic that they are missing.

Introduced by Kent Beck in 2000[30], Continuous Integration aims to mitigate such problem thanks to the fast pace of code growth. As indicated in picture 2.4 focus of CI is on the first step of the programming lifecycle, covering the important

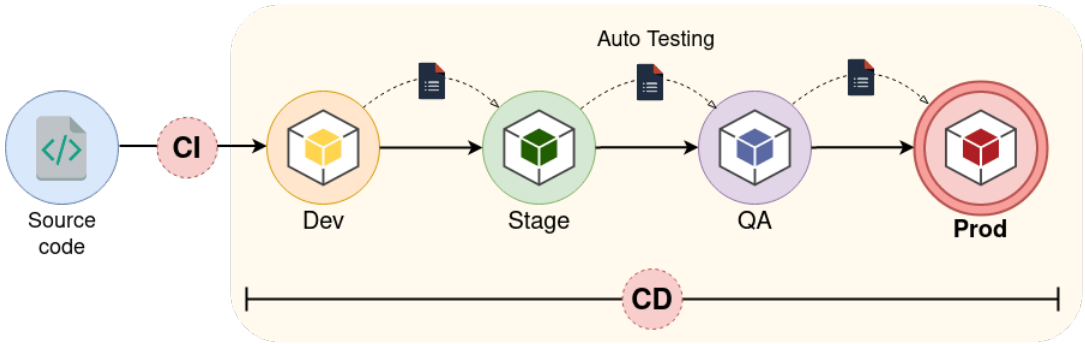

Figure 2.4: CI/CD integration in the code lifecycle[28].

step that goes from the coding phase to the build. The key concept is quite easy as it is effective: frequent integration of the code allows developers to work on an updated code base, therefore drastically reducing the likelihood of conflicts. This won't totally prevent clashes, but it will surely reduce the number and, most importantly, the complexity of the required fix. Less developers will be involved in the case and probably they will also have to concentrate on small snippets of code which have been written recently (probably within one or two days). This pattern is in line with the Beck's mantra, which states:

> "If it hurts, do it often and it will hurt less."

The CI principle is not only aiming to integrate the code, but it is also focused on the automation of building and testing new commits. Using pipelines (described in 2.4.3) to automate the build will enable the developer to get notifications just in case something went wrong and, eventually, quickly fixing the problem. Moreover, constantly having an updated and working version of the software will open more possibilities for testing team to accomplish their task more efficiently.

Continuous Deployment (CD)

If the CI is focused on the early stage of the code lifecycle, *Continuous deployment* wants to follow the rest of the chain. The goal is to push as quickly as possible the

code into production through small and frequent changes. Perfectly following the first principle of the Agile Manifesto[4], CD wants to *satisfy customer through early and continuous delivery of valuable software.*

If the first step of the process is covered by CI, which translates code into software with the build, CD wants to push the software into different steps represented by the environments to finally hit the production. Different tools may be used in order to automate the deployment of the software through the different stages, even if the purpose is always to describe and carry out movements of the build across environments. Order of deploy, timing a versioning/tagging need to be discussed within the organization and then translated into a real configuration for such auto deploy tools.

Lastly, depending on the organization and its nature, *decision making* can be required when in comes to release new versions in production. As already mention in section 2.4.1, companies may target the production release as a risky step that needs to be evaluated each time a specific version is ready to be deployed. Some companies refer to their Change Approval Board (CAB) to take decisions. Surprisingly, CD acronym is sometimes overloaded: when the decision step is required, *continuous delivery* terminology should be used rather than deployment. Unfortunately, often both of this nomenclatures are associated with the same abbreviation, confusing relative concepts as well.

It is important to notice that opting for *Continuous Delivery* may be the deterrent used to hide the fear of adopting a complete Continuous Deployment ethic. As stated in the Humble's book[28], sometimes fear takes over rational decisions, leading project managers to the feeling of losing the control over what is delivered to the customer. Even though this may sound reasonable, it is also true that the risk is tremendously lowered by the minimum amount of change that each small release will bring. Furthermore, as discussed in section 2.4.1, *Canary releases* enable

restricting potential damages as well as collecting data on how customer perceived the newer updates.

A great instance of successful continuous delivery implementation is the Paddy Power[31]'s one that L. Chen described. The organization went into a massive shift during 2014 from traditional software development model into a more automated reality. At that time, the company counted 4000 employees in the technology department, with teams composed by two up to 26 developers, depending on the entity of the project. Before the big change, the company used to release just 6 version of the software during one year. After the shift, Chen noticed the increase in the quality of the delivered product, obviously related to the boost in customer satisfaction: getting frequent updates drive the customer to a better overall perception of the product and stimulate also to provide feedback that can be implemented and released fast. It is a virtuous cycle that leads to success. Chen pointed out also challenges that have been introduced by the shift: the most important one has been identified as the breakage of barriers between teams and the conceptual adoption of real collaborative work. Organizational challenges are the part of the transaction that may require more effort and time to be defined. Lastly, but not less important, a technical dare has been the creation of a common workflow to put in place CD; there are no standards and this consumes resources in order to pick the correct tools and best practices to be adopted.

As for CI, *pipelines* represent the best mechanism to enable autonomous deployments.

2.4.3 Pipelines

As stated before, *pipelines* are the backbone of every kind of *continuous framework*. Pipelines are the definition of all those automatic steps that need to be accomplished with respect to the CI/CD policies that the team have decided. In other

words, pipelines can be considered as a executable specification of the tasks that an engineer would have to accomplish manually right after the coding period. These specifications are usually split into four consequent steps, where each of them may contains different tasks in turn:

- **Source:** this is the pipeline entrypoint, usually fired with a source commit made by the developer. It can also be fired manually or scheduled, but it is quite uncommon. Usually this step is not explicitly defined since its unique goal is to start the rest of the operations chain;

- **Build:** source code needs to be combined with dependencies in order to create the executable software. This step may be skipped in case the language in use is an interpreted one (such as Python, Javascript, Ruby etc.) but it is necessary for all the ones that need to be compiled, such as *Go, C, C++, Java* and many other. It is particularly important to notice that, in a microservice context, this step is *always* required independently from the programming language, since the container requires to be built;

- **Test:** as soon as a runnable instance of the source code has been produced, the software itself can be tested in order to verify its correctness and expected behaviours. Tests to be run against the build are written by the developers, and may differ in typology. The biggest distinction is between *unit tests* and *integration tests*; the first one is focused on testing a the smaller feature of the software, which should be the more atomic and simple as possible; on the other hand, integration tests or *end-to-end tests* are made from a user perspective, testing multiple consequent interactions with the software that can be made by a user to consume a specific feature. It is important to notice that writing and running more test means to have a more reliable code, even if this phase may take quite a lot of time. In order to mitigate this slow down, a team may

want to execute tests just in one of the environments or reduce the number of assessments in the early stages;

- **Deploy:** actions performed in this phase really depend a lot from the environment and from the policies that have been decided by the team/organization. It is easy to understand that deploy features in DEV environment can be done without risking compared to deploying in production. The tendency is to automatically fire this phase in the first steps of the deployment and fire this phase manually for production deployment.

Whenever a single task contained in one of the previously described steps fails, all the pipeline stops generating a notification. At this point it will be easier for the developer to check and solve the error, since the modification that produced the failure has been made recently. As soon as the problem is solved, the developer can commit again and a new pipeline will be fired. Whatever pipeline tool is in use, it should always provide logs and tracebacks of what has been done so far and error messages that generate the failure state, so that the developer will easily locate the source of the problem.

3 Security best practices in software development

This chapter attempts to frame the current state of the art regarding security processes and their integration in the normal software development flow. Such best practices, taken from literature and from the the most virtuous contexts, represents the starting point for the investigation aims of this study.

Security cannot be taken into account as an additional phase of the normal development, where all checks and verification are applied in order to certify the robustness of the final product: on the contrary, security lays on all different steps of the development adopting different approaches to reach the highest level of effectiveness. It is important to keep in mind the importance of the shift-left paradigm (discussed in section 2.1.2), without losing the focus from the last phases of the process.

In the next sections, current methods defined as the best approaches are grouped by the main phases in which those belong to. Starting from the planning phase, where measures are defined at an higher level, methods become more technical for code, build and test phases.

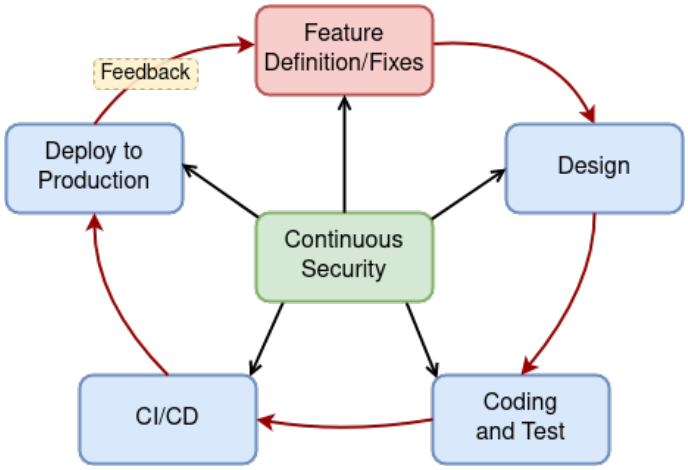

Figure 3.1: DevOps cycle proposed by Jim Bird[32].

3.1 Secure Software Development Lifecycle

A great overview is provided by the concept of *Continuous Security* that Jim Bird tried to frame into the diagram shown in figure 3.1: security is achieved following the SDLC steps that have been discussed in 2.1.1, demonstrating again how security is a central feature.

Even more interesting is the Secure Software Development Lifecycle (SSDLC) framework openly released by the Unity AppSec Team[33], which provides a more practical approach. Despite small differences, concepts that rely behind these two perceptions of security are quite in line. The Unity documentation defines a parallel lifecycle timeline (shown in figure 3.2) to the usual SDLC one, identifying practices to adopt in each block of the process. In table 3.1, each stage of the Unity SSDLC is associated with macro-tasks to be integrated in the normal workflow. The aim is to leverage security over the development process.

The schema proposed by the UnityTech will be adopted in the following sections in order to frame and describe measures that nowadays are considered to be the best practices in infosec. By itself, SSDLC is useless if not associated with an effective set of tools and processes. In fact, activities listed in table 3.1 are high level activities

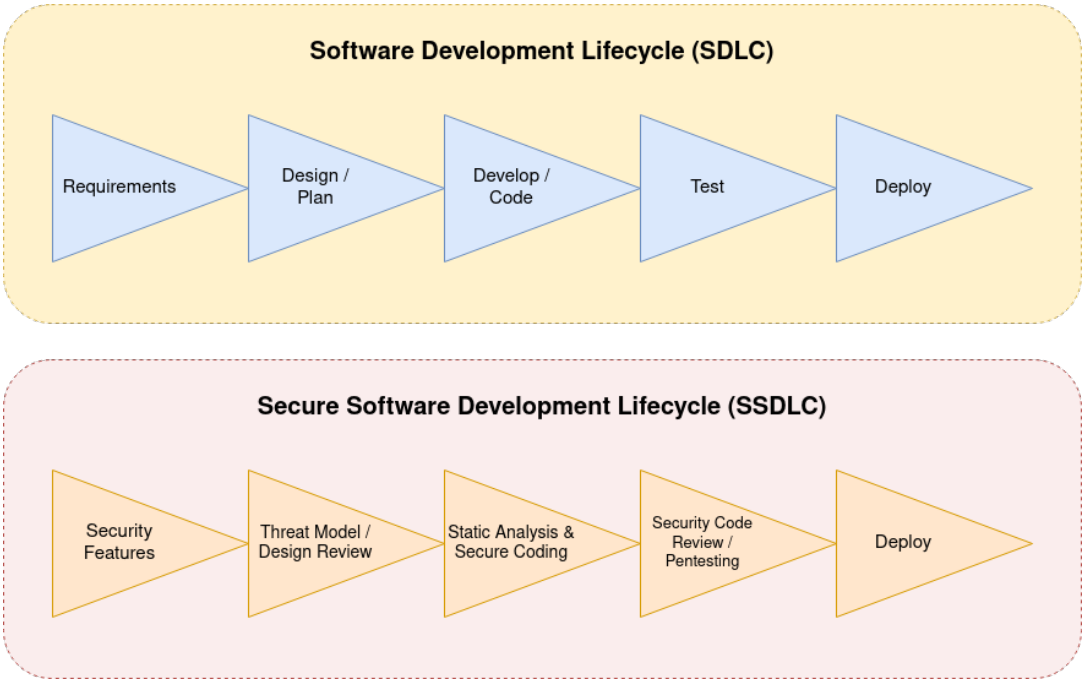

Figure 3.2: Unity SSDLC parallel with respect to normal SDLC[33].

that deserve to be deepened and integrated.

Requirements	**Design**	**Implementation**	**Test**	**Deploy**
Security Requirements	Design Review and Threat Models	Static Analysis and Secure Coding	Open discussion and Security Testing	Pentesting and Incident Response Support

Table 3.1: Unity AppSec Team SSDLC macro activities offering[33].

3.2 Requirements phase

In the earlier stage of a project, the requirements definition needs to evaluate the boundaries and rules that will be applied during the entire development and shared high-level guidelines to be compliant with. Elements to be taken into account are law and specific regulations related to the project topic, even if internal and customer care policies should be involved as well. On top of the assumptions and decisions

that are taken during this phase, security requirements should be involved in the discussion in order to identify standards that need to be observed. The company should have a security policy already in place that can speed up the process, but external standards are usually interesting to be followed to enhance the security level, therefore gaining in reliability and trust from customers. Depending on the field, many security guidelines have been detailed: explicative instances may be represented by Health Insurance Portability and Accountability Act (HIPAA) regarding healthcare or Payment Card Industry Data Security Standard (PCI-DSS) standard for what concerns credit cards. Such *information security standards* are responsible for the mitigation of environment-related threats and are mostly associated with sensitive scopes.

Exception is made for ISO 27000-series standard, which is the most important general security standard. As such, it deserves to be described more in detail in the following section.

3.2.1 ISO/IEC 27000-series standard

The aims of the ISO/IEC 27000 standard family target to compensate the lack of information security in business organizations, whether for small companies or for big enterprises. Sensitive data is the main actor taken into account, which can be identified in financial information, customer data, employee details and internal intellectual property. The delivery is a complete framework of practicalities related to the enhancement of information security measures. As indicated in the name, the standard has been released and currently maintained by the united effort of International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) organizations.

In particular, the ISO 27000 acts as an introduction and background definition for the well known ISO 27001 standard. This latter explains in detail requirements

for an compelling Information Security Management System (ISMS). Having such a standard helps companies to centralize security controls that may be already implemented within the company infrastructure, but not properly used. The result is an efficient and complete ISMS, able to systematically check the identified vulnerable points. As stated by Georg Disterer[34], the ISO 27001 is a signature of reliability for the company who claim it, since it implies meeting strict requirements. Moreover, the certification comes from a third-party international organization such as ISO, consequently increasing both the reputation of the subscribed company and the customer's faith. In addition, the standard is constantly updated in order to match with new technologies and their related threats that come along with, pushing companies to be keep their infrastructure updated and in line with the latest guidelines.

It is important to reference the main objectives of the ISO 27001 standard that Disterer[34] pointed out. Apart from the standard itself, the paper highlight strong concepts to take into account during the requirement definition such as *human resources security* or *information system acquisition, development and management.* Being compliant with all the requirements can be a tough target to be accomplished by companies, but even if that is not possible, the standard should be used to pinpoint the major topics and elements in which to focus on in the requirement phase and, in general, it should always be perceived as a target to hit.

3.3 Design phase

Requirements shape the boundaries of the project, but the foundation of the software architecture need to be crafted just before to start to implement the product. In fact, there is no code involved already in this phase since the focus is on the elements that will be used and how those will be connected together. Even if some decisions may appear to be straight-forward for designers, this is not replicable at all security-wise

speaking. For this reason, all design documentation that will be produced should be reviewed from security perspective and, even more effective, security experts should be involved in the design process to facilitate decisions and highlight weaknesses that may go unnoticed. To help in this process, separate sessions can be organized to review the architecture and find all possible flaws and vulnerabilities of the provided solution. This discussions are usually named as Threat Modeling (TM) sessions; high-level frameworks and software tools has been produced during the time to support such practice, but still today lot of organizations lack in the usage of such an important practice.

3.3.1 Pre-coding

Before get into the process of writing code, some preventive steps should be done to design in the best way the code architecture and safe time and effort to solve problems that, almost certainly, will come out later on.

Third party libraries validation

It is a popular and verified quote which states that *there is no need to reinvent the wheel*. Third party libraries are really important for developers because they allow to speed-up the code creation problem avoiding to spend effort in something that has been already developed, tested and debugged by someone else. Such libraries' source code can be either publicly released as open source resource or private and provided by the third entity as *blackbox software*. Security-wise speaking, it is strongly recommended to prefer the first option, even if the library itself entails a subscription payment. *Whitebox source code* is more secure by design, since anyone can access the codebase, review it and freely start a discussion over possible problems. Moreover, the risk of running code that execute something different with respect to what is stated by the third entity is canceled. Fortunately, as confirmed by the Snyk in

its annual report about security in open source[35], the open source expansion is continuously growing and adopted by almost all the products that nowadays are leading the software development.

Keeping that in mind, it is also meaningful to check the shape of the repository and how it is maintained. Accessing the repositories from Github[1] is always a good practice, since a lot of insights can be detected from there. The most important information to detect is if the library is still actively maintained, and this can be done inspecting the commit history. Moreover, also the list of issues are important to detect some possible major problems which have been detected by users and that are still unresolved. Lastly, it is worth to check the number of contributors that work behind the project: relying on a library maintained by a single person is not recommended, since the project could easily be abandoned and is too dependent from the developer's willing. It is better to rely on libraries developed and maintained by relatively big and important teams that can properly support users.

Fail securely

Setup the environment means also to create the possibility for the developer to test developed features in a protected environment in which there are no big availability constraints. Testing features also in early stages can speedup the entire process, whether if this is done in a company development environment (as discussed in 2.4.1) or locally, in the developers' machine. Therefore, *sand-boxing* the testing environment as much as possible is crucial for the development effectiveness and the security-related concerns as well.

[1] `https://github.com/`

Managing secrets

Hard-coding tokens and passwords in the code, even if just for testing purposes, is one of the bad habits that most of programmers still commit. Even if for local testing purposes this can be perceived as a safe operation, at least until the developers forget about that secret, pushing changes in the repository and therefore exposing the secret publicly. As evidence of the high frequency of such event occurrence, one good source of proof is the **sshgit live**[2] website, where fresh new publicly released usernames, tokens and passwords are collected from new commits on git version control platforms such as GitHub and GitLab.

Fortunately, tools and platforms to properly handling secrets are available and represent the best manner how to safely store and access to credentials, tokens and sensitive files.

When an organization is not able to provide a single solution for secret handling, employees and developers usually adopt different and risky methods to store credentials and tokens, most probably exposing the entire system or part of it to huge dangerous threats. For this reason, the **centralization of secrets** assumes a key role. Converging all secrets into a single place helps administrators to build governance and policies over sensitive information, as well as enabling also the capability to **log and audit access** to sensitive information. In that way, also revoking access to specific resources would be easier and applicable with a better granularity.

One may think that creating a simple database would be enough, but actually this is not the case: with more sophisticated solutions it is possible to provide **Access control capabilities**, enforcing policies that otherwise wouldn't be used with a normal database. Thanks to an Access Control List (ACL), both users and applications are capable of accessing only the relevant information for them.

Another way to increase security with automation is to **dynamically rotate**

[2] `https://shhgit.darkport.co.uk/`

secrets for every different entity/service. User/application will access the credentials in the same way, but under the hood the secret manager will refresh secrets, refreshing them and cleaning up once the (short) Time To Live (TTL) of the secret itself expires. Moreover, since there's no direct access to the secret itself for the user, it will be possible to **generate strong passwords** and at the same time, **prevent reusing** those.

When speaking about secrets, it is also important to use encryption in all the steps, so that the information would be useless even if a leak occurs. In this sense, **encryption as a service** allows to safely handle users credentials and, at the same time, release developers from implementing cryptography by themselves.

3.3.2 Threat modeling

Threat Modeling (TM) is the practice that allows a team to create an abstraction level of their system consequently is examined to discover flaws exploitable by a hypothetical malicious user. As mentioned in [36], TM can be disassembled in three main steps:

1. **Asset identification**, where the most sensitive services and information are pinpointed as primary target to be protected;

2. **Creation of a system overview picture**, clearly placing the assets and defining the data flow as well as the interconnections between the different parties of the system;

3. **Threat identification** on top of the previous step, discovering vulnerabilities that endanger the assets.

Along the years, multiple frameworks to improve efficiency of TM has been released: in [37] N. Shevchenko managed to describe the 12 most important ones, while in [36] a bigger analysis was conducted, taking into account 26 methods. This

proliferation is dependent from small changes that during the time had been made to the main frameworks. Even though the most relevant methods can be reduced to few of them, having such a big pool is still worth since the method should be adopted with respect to many factors. This should push teams to analyze their systems from different perspective with respect to their specific needs and the purpose of the product itself. The available amount of time, the experience of the team, the level of engagement of the stakeholders or the targeted focus (security, privacy, risk) are just some examples of considerations that may influence the decision of the most appropriate methodology.

As remarked in [37], the common advice is to organize the TM session as early as possible in the development process. Be proactive, in this sense, while architectural decision are still under discussion, as stated in section 2.1.2, reduces the number and even the impact of all possible threats that may come up.

For the scope of this thesis, there is no meaning to go through all the main methods adopted by organizations but mentioning STRIDE as the most widely adopted can be useful to understand how a TM framework can be structured. Therefore, also TRIM is taken into account because of its different aim with respect to STRIDE: useful instance of different point of views that a team may want to stick with depending on the specific needs. Following this kind of a guidance, the team won't forget to cover all different main aspects to be protected within a system.

Data Flow Diagram

Data Flow Diagram (DFD) plays an essential role in the TM, representing the most important element where to base all the following assumptions and observations regarding security. Not only STRIDE-based methods require such an artifact, but lot of other famous TM methods such as LINDDUN (Linkability, Identifiability, Non-repudiation, Detectability, Disclosure of information, Unawareness, Non-

compliance)[38] and PASTA (Process for Attack Simulation and Threat Analysis) (reviewed in [37]) but that are not included in this literature.

The main purpose of DFD is offering an overall picture of the system, creating at the same time a good layer of abstraction but keeping a sufficient level of details too. The most important entities that need to be included in the DFD usually are:

- System interactions with external entities,
- Processes,
- Data flows,
- Data stores.

Finding the good ratio between details and abstraction is the most difficult side of creating an effective DFD. Vulnerabilities often rely on tiny small details that consequently could be exploited to threat the entire system. Therefore, omitting security assumptions that have been made while conceiving the system architecture may mislead the TM session itself. In this regard, a detailed analysis has been conducted in Belgium in 2018[39]. L. Sion and his team highlighted how much the lack of security and privacy-related information, as well as for traceability of accomplished decisions, can badly affect the consequent threat discussion.

The proposed solution aims to enrich the base DFD with all the elements that can influence, even if slightly, security and privacy aspects of the architecture. The easiest example is shown in figure 3.3. Drawing 2 entities in communication is worth and can be easily understood, but labelling such connection with HTTPS is way more informative for the audience, which consequently can take for granted that from that connection *integrity*, *confidentiality* and *destination authentication* are enforced because of TLS.

Despite the basic example, many other assumptions can bring to the table important notions to be taken into account: the usage of containers, executing a certain

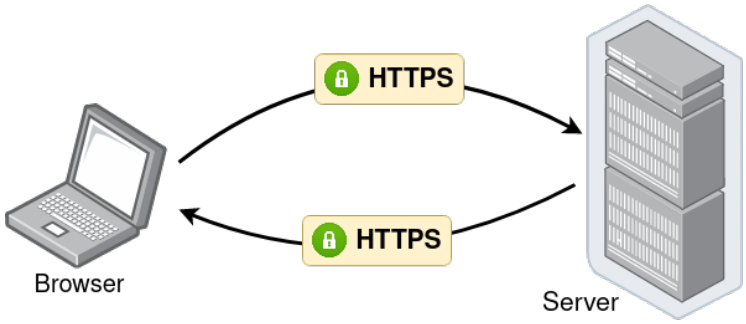

Figure 3.3: DFD section representing an HTTPS communication.

service in a sandboxed environment, OS protections applied and encryption are just few instances that are often used in such a context. These details in the DFD can significantly speed up the TM session, therefore decreasing the number of false positive and false negatives. Moreover, this will increase the attendees' confidence and consequently, the quality of the analysis itself.

STRIDE

Among all different frameworks and methods taken into account in [37] and [36], STRIDE can be easily elected as the most frequently adopted and, therefore, the most mature. For this reason, quite a lot of derivations of this method have been created during the years. STRIDE was created in 1999 and its popularity started to grow since 2002, when Microsoft decided to adopt it. As many other techniques, everything starts from the Data Flow Diagram. The success of the session strongly depends from this step, since it represents a common baseline of knowledge for all the attendees of the session used to discover new threats in the second phase. In this stage, the acronym of the name itself is used as a mnemonic reference to take into account aspects of the architecture.

Within the name of the technique, the following threats are carried:

- ***S**poofing*: threaten *authentication* when an attacker claims to be someone or something else to gain privileges;

- ***T**ampering with data*: disrupt the *integrity* of information contained in memory storage;
- ***R**epudiation*: break the *non-repudiation* principle through the possibility of renouncing any taken action, whether the subject is honest or not;
- ***I**nformation disclosure*: leaking information to entities that shouldn't have access to it, breaking the *confidentiality* principle;
- ***D**enial of Service*: causing a service breakage, undermining the *availability* of a provided service;
- ***E**levation of privilege*: an entity is able to perform actions which should require an higher set of privileges with respect to the currently owned, escaping *authorization* enforcement.

In support of the acronym itself, checklists and collections of questions have been written to help highlighting possible vulnerabilities.

This method is usually criticized because its tendency to be time-consuming, as well as the attitude to generate quite a lot *false negative* cases.

TRIM

Despite TRIM sharing the general idea with STRIDE, this framework is completely centered in providing a guideline in order to evaluate privacy concerns. The aim is to capture possible design mistakes in the system which could undermine personal data. The concepts delivered with such acronym are all about data handling and protection:

- ***T**ransfer of personal data*: confidentiality over the transfer and legal constraints;

- ***R**etention and **R**emoval of personal data*: definition of a specific life-cycle for personal data storage and the possibility to completely delete those;
- ***I**nterconnections and **I**nference*: destroying pseudonyms and strong correlations between personal data items;
- ***M**inimisation of personal data*: reduce the stored information to the minimal amount necessary for technical requirements.

Despite some letters are shared with STRIDE, it is easy to notice that these two methods are quite far from one another, offering a different perspective. It is a good practise to combine both methods in order to get the best result out of the TM session.

Threat modelling applicability with Agile

On top of all the considerations that have been done so far, it is clear that threat modelling has a big relevance within the process of development of whatever project. Futhermore, integrating such process into the normal Software Development Life-cycle is a step that needs to be carefully carried out to transform TM into effective tasks to be accomplished in order to enhance security. As pointed out in [37], TM is completely applicable with Agile development methodologies[3] and can be adopted with respect to the sprint timeframes. Despite this compatibility, such integration is still a delicate step that brings some challenges to be faced. In [36], four different Agile-based Norwegian organizations, where TM sessions are part of the SDLC, have been taken under analysis. The study pointed out the difficulty of perceiving security as a top priority. Employees were interviewed and what came of can be summarized in a brief list of challenges that reduce the efficiency and the positive effects of the TM itself:

1. The *lack of motivation* is quite common in three out of four companies because

of the small relevance conferred to security in general;

2. Often employees reported the *difficulty in identifying threats*, which usually is derived from the lack of experience or from the decision making required when an identified threat need to be labelled as relevant or not with respect to the system context;

3. 75% of companies perceive TM as a *time-consuming task*, which can be the reality if the session is not facilitated by some experienced figures and if DFD is not complete or is even missing at the very beginning of the session;

4. Almost all the interviewees mentioned to have difficulties in the *"definition of done"* when it comes to evaluate steps to be accomplished and checkpoints to consider a threat as completely mitigated.

The investigation carried out by K. Bernsmed and M. G. Jaatun also defined three of the *best practices* that influx positively in the TM practices:

1. Even if not particularly relevant for assets definition, *developers need to be involved in the TM* to bring relevant and practical aspects to the table, maturing in them a critical spirit with regard to security;

2. The *usage of checklists* is a simple as effective method to clearly define the process, simultaneously avoiding wasted time and forgetting discussion points;

3. A *regular scheduling* of the sessions keeps the team aware, helps to completely integrate TM into the SDLC and therefore reduces the needed effort per each session.

Tools

In support of the long list of TM methods, some tools have been developed by big organizations such as Open Web Application Security Project (OWASP) in order to

be integrated into their processes. Consequently, these have been released to help other companies to speed up and increase the effectiveness of the sessions. These utilities don't have the pretensions of replacing a complete framework, but to support the latter, making the meetings more dynamic and less heavy. The following list is a brief collection of the four most widely used proposals:

- OWASP - Threat Dragon
- Microsoft - Threat Modelling
- OWASP - Cornucopia
- Mozilla - Seasponge

Of course, adopting any of these utilities is not required, especially in such teams where a good level of experience is already achieved. On the other hand, such tools may help where TM is not a consolidated practice.

3.4 Implementation phase

Developers are the main actors in this step. For this reason, they can also be the biggest threat: they tend to focus on solving the issue that they have been assigned to and usually they are pushed to accomplish the task in the fastest way as possible. Unfortunately, this leaves security in the shadow and that is why it is so important to invest on the creation of a real sense of *security awareness* in the developers. The target to reach for a company is to count on developers to be mature enough to critically think about the code that has been written and relate it to a possible threat. For this reason, it is so important for a developer to memorize and digest the OWASP Top 10 vulnerabilities[40], that, as stated by OWASP itself, is

> "Globally recognized by developers as the first step towards more secure coding."

In this way, this collection of threats won't be anymore a list to check every time at the end of a coding session, but instead a collection of concepts unconsciously and constantly recalled while writing the code.

That being said, still a lot of details and precautions can be take in place to push further the effectiveness of *secure coding*.

3.4.1 Coding

In the process of coding, the developer can still take into account some aspects that can decrease the likelihood of introducing flaws. The first step to write secure code is studying. Being able to consider threats while coding requires the developer to be aware about common practicalities and problems that may undermine the security of the overall project. But on top of that, few more points can be highlighted in order to further help the developer.

Linters

The usage of *linters* may help to anticipate security-related problems even before the code is committed in the repository. Those can also be used to help the developer to improve his coding style, learning new patterns and techniques to avoid flaws. Usually such tools can be installed as an extension for common IDEs, allowing the programming environment itself to provide hints and pinpoint possible flaws in realtime, while the developer is in the process of writing. Usually a linter is strictly linked to a specific programming language so that specific language-related problems can be detected more precisely.

A linter can be defined as a software able to run *static analysis* of the code, therefore such tools can be also integrated in the CI/CD pipelines (discussed in 2.4.2) or simply fired before committing the code.

The tool is particularly important when used with a team vision: a linter can

help to maintain a heterogeneous knowledge within a team. Juniors can see liters as a senior member that gives guidelines and facilitates the growth process. Seniors can enforce a consistent style on a team.

Language best practices

To integrate the service that linters are already providing, the developer may want to go through some documentation that can provide a list of best practices and practicalities to be adopted while coding with a specific programming language. In fact syntax, design or constraints of a specific language may open the side to different problems which usually are well known and can be easily mitigated during the writing process. The collection of best practices for *Go*, *Node.js*, *Ruby* and *C#* provided in the UnityTech SSDLC framework (proposed in 3.1) is a good example of reference, but many other guidelines provided by referenced authorities such as OWASP can be found for the most common programming languages.

Sanitization

Lastly, it is important to bear in mind to always verify any kind of input with sanitization methods, as well as checking the provided outputs. Cross-site Scripting (XSS) attacks are one of the most common vulnerabilities since they rely on programmers negligence and are quite difficult to detect. One single defect may represent a potential huge risk for the entire system.

3.4.2 HTTP security headers

More often enterprise sales relies on a new sort of benchmark assessing a security evaluation over a company or a product. For this reason, services that provide that sort of *security scorecards* are rising in popularity. Because their relevance is ramping up, it becomes to be important to understand where their score is based

on. Based on the scoring methodology[41] applied by Mozilla Observatory[3], which is one of the most famous services in this field, such score is assigned based on two main factors. *IP reputation* is one of these: it always refers to blacklists or spam lists where bad nodes are collected. As far as the company taken into account is not spamming or frequently affected by malware infections, this factor shouldn't effect negatively the overall score.

The second and most relevant factor to really take into account is the *HTTP security headers* used during the communication. Since the IP reputation is quite easy to be achieved, security headers set on public websites determine the majority of the score outcome. Headers can directly address specific problems, mitigating different possible threats. Therefore, it is meaningful to group and describe them by purpose, trying to provide a better understanding about where and how headers can be relevant for mitigation.

A complete overview about all headers which are related to security is provided by OWASP in their *OWASP Secure Headers Project*[42].

Cross-origin Resource Sharing

One of the most powerful features that has revolutionized web applications in general has been the possibility to link resources to allow communication in between different parties. Despite the improvements carried out by the feature, such feature opens the application to many possible attacks. Restrictions over the shared data between parties are needed, and therefore an origin based security model has been created in order to regulate communications. Even with the security model put in place, it is particularly difficult to provide a solution which can fit all possibilities. In fact, Cross-Origin Resource Sharing (CORS) setting are very endpoint dependent. Despite such preamble, there is still room for general guidance which can be helpful

[3]Mozilla Observatory online service

to provide a reasonable usage of CORS headers.

CORS can be considered as a whitelisting tool for Same Origin Policy (SOP) mechanism. For this reason, the usage of wildcard `*` is strongly discouraged in allow-headers (such as *Access-Control-Allow-Origin* or *Access-Control-Allow-Methods*). The only situation in which such header can be used is when a publicly open app/APIs needs to be served.

Lastly, it is worth to bear in mind that CORS policies completely differ from measures that can be taken into account for Cross Site Request Forgery (CSRF) attack mitigation. In fact, SOP may help only when trying to prevent any token to be read form a different malicious domain, which is not a target of a CSRF attacker. The attacker's aim is to let the user perform the malicious request performing a client-side attack and third party origins are not really involved in such mechanism.

Content Security Policy

The purpose of Content Security Policy (CSP) is to mitigate XSS attacks by restricting resources loaded in the browser. JavaScript, CSS, fonts, images are just intances of what can be controlled by this header. Each different kind of resource can be treated in a different way. By default, if not specified in the directive, all resources are allowed to be loaded. As general guidance, the stricter the directive, the better it is for security of the web application. Therefore, blocking all the unnecessary resources specifying the policy as: `'none'` is the preferred choice. It is still worth to define policy to `'self'` in case the domain has only trusted contents.

Many tools are available for different browsers in order to create CSP drafts and for checking them as well.

An example of CSP header can be the following:

```
Content-Security-Policy: default-src 'self'; img-src 'self';
object-src 'none'; script-src 'self'; style-src 'self';
```

```
frame-ancestors 'self'; base-uri 'self'; form-action 'self';
```

In this case the following directives are enforced:

- Default to only allow content from the current site;
- Do not allow objects such as Flash and Java;
- Only allow images, scripts, styles, frames and forms to submit from the current site;
- Restrict URL's in the tag to current site.

Cookies

Cookies are particularly interesting for an attacker in order to achieve unauthorized access over protected resources. Therefore, is particularly important to enforce some security constraints over the usage of such powerful but potentially dangerous tool. In the response headers it is possible to specify policies to be applied with cookies.

The *Secure* directive is important since it enforces the cookie to be transmitted exclusively via encrypted connection (HTTPS required).

Whenever possible, it is also strongly recommended to use the *HttpOnly* directive to deny the possibility to read the cookie using JavaScript. *SameSite* directive could be used to forbid the cookie to be sent via cross-origin requests (assigning *Strict* value), or at least to limit it (using *Lax*). This feature is particularly useful for CSRF attack mitigation.

It is also recommended to apply the shortest expiration date as possible and assigning the some restricted path as well to limit visibility.

Information leakage avoidance

Sometimes also removing some headers may help to restrict attack surface. It is the case of headers such as *Server* or *X-Powered-By* headers, where information about

the backend software and version are leaked. Usually an attacker aims to know this information in order to look for possible public vulnerabilities related to the specific setup in place in the server. Therefore, is recommended to simply avoid to include this headers in responses.

Cache control

As for useless headers that may leak relevant information for an attacker, cache can represent a target for an attacker who is willing to retrieve personal information. It is a good practice to avoid caching pages where sensitive information is contained, reserving this feature only for static resources such as images, CSS files or similar. *Cache-Control* header can be included in the responses to force the client to not use cache for specific sensitive data. *Pragma* header has the same purpose and is used for backwards compatibility with HTTP/1.0 caches.

Don't roll your own crypto

As general advice, it is always a bad idea to start implementing customized cryptography algorithms since these require an extensive knowledge in mathematics and need to be tested for long time. Therefore, relying on well-known and up to date algorithms is the best choice, which reduces development time and increase security.

Moreover, keeping in mind how and why the used protocols have been implemented will help the developer to correctly apply such protocols. Therefore, studying at least the basic mechanisms behind adopted protocols is strongly suggested.

3.5 Test phase

Now that the code has been committed, fulfilling the previously defined requirements is needed. Thus, the source needs to be evaluated from different perspective, trying to fire the highest number of tests that can be accomplished autonomously.

There is still room for manual testing of the application in this phase, but such tasks should be limited to few restricted tasks; reviewing logs of automated tests is one of those; *exploratory testing* together with manual *penetration testing* should be constantly accomplished in order to look for new vulnerabilities and threats may not have been caught by automatic testing.

Fuzz testing (or *fuzzing*) is another great way to discover implementation bugs and possible malformed payloads that may disrupt the target service. This black box technique aims to automatically detect problems through the generation of a multitude of requests with randomly generated (and therefore also malformed) inputs that may have been left uncovered by developers. The technique, firstly presented in 1989 by Professor Miller from the University of Wisconsin-Madison[43], nowadays can be performed by many tools that usually are particularly used in penetration testing. Fuzzing can be considered as a specific kind of test that can be placed in between manual and automatic testing, since the point of injection for random data should be carefully and manually identified.

Automation of tests includes different kind of checks. Firstly, it is worth to constantly monitor performance of the resources involved in serving provided services, either virtual or physical ones. Most probably, resources that are going outside boundaries of normal consumption are the sentry of problems. Having good notification system may avoid brutal attacks such as a Denial of Service.

Ensuring that security standards are also doable with reasonable amount of effort: analyzing security headers and TLS configuration is an easy and cheap way to do so.

In the early 2020, it has been reported by the Snyk State of Open Source Security Report[35] that more than 75% of all code vulnerabilities reside in indirect dependencies. This should be enough to convince the reader about the importance of analyzing dependencies that are involved in the application. In this sense, mi-

croservices may increase complexity since each service is potentially running with a different programming language and platform. Therefore, it is difficult to identify a simple solution able to solve these checks. Each organization needs to craft their own implementation depending on the architecture in place.

Going deeper into the code inspection, *smoke tests* are an essential element of this phase. Main functions are reproduced by an automatic tool that is able to replicate an user that tries to use every single feature that the application offers. The aim is to certify the correct behaviour of the application with respect to the provided inputs.

Smoke tests can highlight possible vulnerabilities exploitable from discovered errors, but more specific and security-centered tests can be written as well. Black-box approach can be used as well to replicate possible behaviours that an attacker would try to perform in order to penetrate into the application. This testing technique is called Dynamic Application Security Testing (DAST) and it may be useful to catch authentication or session issues, memory leaks or other weaknesses of the system. This technique tends to produce quite a lot of false positives and it is usually integrated with a good bug tracking platform to decrease the rate of misdetections. Moreover, DAST requires to be run in a testing environment where malicious payloads can be pushed without risking corruption of sensitive data.

Static Application Security Testing (SAST) is another kind of investigation which can be conducted; differently from the previous one, whitebox approach is followed allowing a security-centered inspection of the source code in order to find specific dangerous snippets. Those well-known patterns are mostly language dependant, therefore SAST tools are usually integrated into linters (described in 3.4.1).

3.6 Deploy phase

A released product includes also assistance to guide the customers and help them when problems occur. Maintaining a product can be defined in different manners from assisting end users or feed the developers' backlog, patching or improving the project.

The most direct and effective way to predict the future direction of a product is often considered to be represented by feedback of customers and end users. To do so, proper methods should be designed in order to process in the best way feedback. Each suggestion need to be translated into descriptive task(s) for the development team. The purpose of feedback processing is to close the development cycle, feeding back the design phase.

As already hinted in 2.4.2 section, it is extremely useful to gather insights about the usage of the product. Such information can be considered as an implicit feedback generated by customers that actively use the provided platform.

Insights analysis does not include only statistics over the usage of the platform, but involves monitoring resources used to provide the service itself as well. It is usually the DevOps' prerogative to setup an effective alerting system that can really help to promptly detect problems that have to be solved as soon as possible. Traditionally, alerts are delivered to developers and DevOps using *email*; despite this tendency, the raising usage of dedicated *instant messaging platforms* contributed to push the development of bots able to directly post alerts into chats. The implementation of such feature is not complicated itself, since the majority of chats already come with the possibility of bot integration (whether as native function or through the usage of third party plugins). Actually, the most difficult part is to fine tune the occurrences of messages in order to avoid the generation of noise in chats. Thus, providing too much alerts will increase the likelihood of false positives, therefore discouraging developers to take seriously such alarms. At the same time, restricting

too much the triggering range will probably generate false negatives. This responsibility is assigned to DevOps, who have to define a reasonable threshold, as well as correctly addressing alerts only to the relevant personnel.

The best way to handle incidents that may happen during the lifetime of the deployed software is to document procedures that should be adopted in such cases. When an emergency occurs it is difficult to take meditated decisions, therefore it is desirable to structure procedures with a cool head to speed-up and improve incident recovery actions.

As stated in [44], the practice of *crowdsourcing* information security has been taken more and more popularity within organization. *Bug Bounty Programs* are one of the most representative examples of such phenomenon. Organizations may want to outsource penetration testing of their products to professional testers, offering a reward in case of discovered and well-reported vulnerability.

Lastly, another good practice for InfoSec figures, developers and DevOps is to always keep an eye over *cybersecurity-related news* to be always updated over new discovered *vulnerabilities* and *zero-days*[4].

3.7 Retirement

In a future the product that it has been developed will face and end and it will be retired. Unfortunately, the retirement of a project is not always well treated in companies and no guidelines are defined. Despite this negligence, implications of the retirement should be considered and handled in a proper manner. In the worst scenario, failing to dispose properly the resources that have been used so far may also lead to information leakages.

That being said, it is important to follow a good *migration plan* in case a new

[4]A zero-day vulnerability is a flaw that is unknown or still not mitigated by the vendor of the software.

product will take the place of the old one. At the same time, all the collected and sensible data should be archived properly to avoid leakages and legal implications related to such information. All resources that are not supposed to be used anymore, whether physical or virtual ones, should be destroyed. Lastly, it could be also important to clean up DNS names in order to avoid future collisions or takeovers.

4 Cutting-edge approaches for secure development

After the analysis made in chapter 3 about how development teams try their best at integrating security into the development process, research has been performed, trying to identify possible cracks and sanitize them with new approaches. This chapter provides practical tools and processes to adopt in order to improve the security analysis in the entire development process. In section 4.2, proposals can be adopted in every development environment, while in 4.1 more microservice-related techniques are documented.

4.1 Improvements for microservice environments

Container clustering orchestration has become very popular whenever containerized services are in use. According to the 2020 Snyk report[35], Kubernetes has been chosen by the 44% of the interviewees. Despite the wide adoption, the survey pointed out how users tend to forget about security requirements. In fact, 30% of the population confessed to not revising K8s manifests at all.

These statistics should be enough to underline the importance of identify specific tools to help monitoring new flaws in container clusters as well as enhancing the security level with new measures.

4.1.1 Kubernetes security management

Kubernetes take into account security by default, efficiently handling IP addresses of the pods contained in the cluster and applying controls in order to ensure a safe communication in between different nodes. Despite these important features, the orchestrator provides only basic security measures, leaving room for advanced security monitoring and compliance enforcement.

Thus, admins and DevOps can count on a set of specific tools which have been designed for the purpose. Among many good products which solve this demand, the Aqua Security[1] solutions are recommended choice of the Center of Internet Security (CIS)[2].

The Aqua *kube-bench* tool[3] strictly follows all the checks that are listed by the CIS Kubernetes Benchmark[45]. The tool, written in Go and provided as a container, is able to discover configuration errors, authorization and authenticaton issues and checks that data flow is reasonably encrypted.

On the same line, Aqua released *kube-hunter*[4] as well. This one can be considered as an extension of kube-bench, which enhance the the effectiveness of the analysis by enhancing discovery and penetration testing capabilities.

Using such solutions is very important nowadays since K8s package managers (such as Helm[5]) are ramping up in their usage. These tools are particularly useful for versioning, but they can also introduce risks that comes from the usage of third party images. This assumption is confirmed by 2019 Snyk Helm Report[46], which states that 68% of stable Helm Charts (Helm packages) contain an image with a high severity vulnerability.

[1] Aqua Security: `https://www.aquasec.com/`

[2] `https://www.cisecurity.org/`

[3] kube-bench Github Repository: `https://github.com/aquasecurity/kube-bench`

[4] kube-hunter Github Repository: `https://github.com/aquasecurity/kube-hunter`

[5] Helm: `https://helm.sh/`

4.1.2 Periodic container scans

As the beating heart of each service of the cluster, the (Docker) container cannot be forget. In the 2020 Snyk report[35], authors highlight that many vulnerabilities have been discovered from container images labeled as `latest`. This means that even using well-known images as baseline for custom containers may represent a potential issue.

Therefore, this leads to understand the importance of a *periodic and automatic container scanning*. DevSecOps engineers should prioritize this practice using open source tools that accomplish the task. Currently, the most well known one is Clair[6], which is able to perform static analysis based of known vulnerability signatures stored in a up-to-date database.

4.1.3 Hacking testing environment

Getting hands on practical examples of vulnerabilities can be really useful and instructive for both developers and security practicalities. For this reason, it may be an optimal idea to craft a sort of sandboxed and vulnerable environment where to train and experiment with exploitation skills. Such playground can be easily created thanks to projects that have been ad-hoc developed for this reason.

The most famous is probably WebGoat[7]: a vulnerable web application which challenges the user thorugh the exploitation of different kind of security holes.

Recently, KubernetesGoat[8] has been released on GitHub with almost the same aim. As the name suggests, a vulnerable K8s cluster configuration is provided to offer a playground where to practise Kubernetes security.

[6]Clair Github Repository: `https://github.com/quay/clair`

[7]WebGoat GitHub repository: `https://github.com/WebGoat`

[8]KubernetesGoat GitHub repository: `https://github.com/madhuakula/kubernetes-goat`

ServerlessGoat[9] should also be mentioned. The tool is provided by OWASP and therefore all the vulnerabilities that are listed in the Puresec ten critical risks in serverless configurations[47] can be reproduced and tested.

Maintaining such testing environments, even combining the mentioned tools or some other resource, can be really useful to provide test bench for penetration testers: there it would be possible to safely replicate issues as well as using the environment as tool for increasing awareness for developers or DevOps.

4.2 General proposals of improvement

Although this work is more focused on analyzing microservice environments, in this section are collected improvements that can be applied to any development environment to increase security. Some of the proposals are quite technical and involve the usage of new tools, but there is room also for more high-level ones that just aim to increase security awareness.

4.2.1 Periodic vulnerability scans

Every organization should be aware of the critical components that may represent a potential intrusion vector for an attacker.

As stated by the Certified Information Systems Security Professional (CISSP) Study Guide[48], each company should establish a routine in order to continuously audit their systems and network. Such process can be either manual or, preferably, automatically scheduled and fired.

Many open source tools have been released as well as other commercial solutions. Each of them can have a narrower or wider scope, respectively resulting in a more accurate or general reporting system.

[9]ServerlessGoat GitHub repository: `https://github.com/OWASP/Serverless-Goat`

Nessus[10] by Tenable is probably the most famous commercial solution available on the market. On the other hand, OpenVAS[11] represents the open source alternative. With a more specific scope, even *Nmap*[12] can be seen as a vulnerability tester due to its advanced network scanning capabilities.

Vulnerability scanners usually are focused on code analysis, vulnerability auditing, network scanning or server/cloud configuration analysis. The most sophisticated ones can also combine more than one scope, even if sometimes organizations prefer to adopt different and more specified scanners.

Another distinction can be made between *external* an *internal* scanners: the first type ground their analysis only on the publicly exposed resources, while the second one has the capability to expand the investigation also to resources that require privileges to be accessed.

The CISSP Study Guide[48] also highlights how crucial it is for organizations that use these tools to bear in mind the link between the discovered vulnerability and the real open threat in the current scenario. Missing such connection may lead to underestimate a vulnerability or, on the contrary, spend too many resources is something which is already mitigated by design of the actual configuration.

4.2.2 Interactive Application Security Testing

In the previous chapter, DAST and SAST techniques have been described as good solutions for vulnerability detections. Despite their importance, both of them still have their limits. Interactive Application Security Testing (IAST) aims to create a hybrid solution which can combine strengths of SAST and DAST, solving at the same time their relative drawbacks.

[10]Tenable Nessus: `https://www.tenable.com/products/nessus`

[11]OpenVAS Github Repository: `https://github.com/greenbone/openvas`

[12]Nmap: `https://nmap.org/`

The analysis is performed by injecting agents into the web application. Such sensors are able to monitor reactions of the app with respect to received inputs (either automated or manual ones). An analyzer processes the detected information and raises alerts in case vulnerabilities are detected. Agents are able to capture configurations, network traffic and external service calls as well as monitoring the overall data flow within the app. Thanks to this capability, more vulnerabilities can be discovered and the rate of detected false positive or negatives can be drastically decreased. In fact, such additional information is used to verify vulnerabilities detected though normal SAST or DAST detection processes. Since IAST solutions have access to the source code, a more accurate report of the discoveries can be achieved, pointing the precise location of the vulnerability in the code. As drawback, this hybrid solution is always more expensive, demanding more resources and a complex installation process, due to agents to be setup.

4.2.3 Code review exercising

Despite many different ways to discover possible vulnerabilities have been developed among the years, reducing the likelihood of generating security holes while coding is still an challenge. This is the conception that the founders of SecureCodeWarrior[13] are pursuing: the provided platform offers the possibility to train developers to recognize inconsistencies, well known anti-patterns and vulnerabilities that may affect the code. The developers of the companies who decide to subscribe with the platform will get access to internal tournaments, learning material and challenges that stimulate the critical thinking with respect to the source code. The challenging formula can be really beneficial for end users, while earned data will offer important insights about the security awareness of the development team.

Moreover, code review process should be integrated in the normal development

[13]SecureCodeWarrior: `https://securecodewarrior.com`

process, usually through merge request mechanisms that enforce a minimum number of approval before being accomplished. Gerrit[14] is an instance of tool which can be applied on top of normal git version control to expand review mechanisms.

4.2.4 Penetration testing automation

Trying to break into a system usually requires capabilities that go outside from what can be automated by a software. Pentesters usually take advantage of their intuition and lateral thinking, skills that are human prerogatives. Thanks to these competences, they are able to find a way to compromise a system and, accordingly, propose a method how to harden the environment in case of attack.

On the other hand, lot of the tasks that may be necessary for information retrieval or so can be automated to facilitate and speed up the pentester process. If a computer is still not able to properly react to certain findings, it is able to highlight results which may be useful for the tester. For instance, fuzzing or enumeration techniques in general are processes that requires time to be accomplished. Such techniques aim to find a valid corrupted payload that may exploit a vulnerabilities, generating therefore thousands useless requests and, in case, just few valuable outcomes. This kind of time-consuming tasks can be easily be automated, periodically repeated and only in case of discoveries an alert can be generated for the tester.

Moreover, not only penetration testers can take advantage of the automation. *Bug bounty hunters* may have the possibility to speed up their workflow in an environment where sometimes be faster than the others make the difference. DevOps may want to let such test run periodically in order to monitor some critical online assets or simply to assert that the web application that the development team is working on is compliant with security standards. Thus, it is worth to integrate such tests into CI/CD pipelines or to recurrently schedule those.

[14]Gerrit Code Review: `https://www.gerritcodereview.com/`

4.2.5 Dashboards

Sometimes, reading through the lines of logs may appear to be confusing. Even though that is not true when correct tools are used in the analysis, offering an immediate and self-explanatory visualization is usually very valuable for a team. The usage of *dashboards* is useful for many aspects of the development such as the resource consumption or APIs call statistics.

The same is applicable for security matters: being able to raise alerts in case of too many wrong login attempts, detect strange connections with malformed payloads or similar is extremely beneficial for security analysts. As demonstrated by Mullis and his colleagues[49], dashboards are a fundamental tool for cyber situational awareness and, therefore, an indispensable tool within every Security Operations Center (SOC).

But the effectiveness of dashboards are not strictly limited to SOCs: DevOps can create custom dashboards highlighting insights from the data and logs taken by the system, helping either developers and security specialists in they job. Such task can be facilitated by the adoption of the measure proposed in 4.2.6.

The most famous products that are used in the creation of dashboards are Kibana[15] (if an Elastic stack is in place) or Grafana[16] as open source alternative. Intermediate solutions can be adopted in order to extract insights from raw logs and data: Prometheus[17] is a perfect instance that allows to create metrics and generate alerts accordingly.

[15]Kibana: `https://www.elastic.co/kibana`

[16]Grafana Github Repository: `https://github.com/grafana/grafana`

[17]Prometheus: `https://prometheus.io/`

4.2.6 Centralized Log Management

Logs are a the most important element which represents the baseline for every kind of analysis. The more the system is complex, the less intuitive is to get useful information out of logs. In fact, every entity included in the system itself tend to produce its own log trace.

Being able to create a common space where all the logs can be directed will offer to the admins a more comprehensive overview and, therefore, the possibility to get more useful insights over suspicious patterns.

Such capability of centralize data is not easy to achieve in every environment. Fortunately, the usage of Server as a Service (SaaS) (discussed in 2.2) helps simplify the task. Amazon Web Services (AWS)[18] gives the possibility to users to convey logs from different regions and accounts. Third party solutions are also available on the market: Logstash is the best choice if an Elastic stack is used, otherwise GrayLog[19] or FluentD[20] are great solutions too. This kind of tools provide an all-in-one solution, offering also a visualization of the collected data. Views can be customized, but most of the times are not flexible enough to satisfy admin needs. Thus, it is important to choose the correct tool with respect to the current system configuration.

[18]AWS Centralized Logging: `https://aws.amazon.com/solutions/implementations/centralized-logging/`

[19]GreyLog GitHub repositories: `https://github.com/Graylog2`

[20]FluentD GitHub repository: `https://github.com/fluent/fluentd`

5 Currently adopted security processes and methods

Despite in the last years security is gaining more and more attention, it is still difficult to identify companies which spent the correct amount of resources in ensuring their systems and internal processes. Unfortunately, security is more often seen as a deplorable task that drains resources and provides no tangible results. The reality is that there is no possibility to achieve a good level of security thinking about it as a patch which can be applied whenever resources are available. As widely discussed in chapter 3, security is something that needs to be taken into account from the very beginning in the design phase, therefore leading to an effective and even cheaper ecosystem that works in synergy with the rest of the organization's parts.

In support of the above, the 2020 Snyk report[35] underline that security awareness seems to follow a growing trend within companies, but technical improvements are still missing in practise. In the same way, GoSecure[50] detected a significant gap in between the participants' perceived security and the reality while confronting the survey's results with the penetration testers' experience. Indeed, 26% of the 2020 Skyn report[35] responses stated to completely lack in security practices.

On the base of this assumptions, the aim of this chapter is to underline how far reality is far from the theoretical model proposed in chapter 3. To better notice the detachment with expectations, the analysis tries to follow the main SDLC structure.

5.1 Requirements phase comparison

While discussing about requirements in section 3.2, the *ISO 27001* standard has been reviewed. After a decade is passed from its publication, an analysis over the adoption has been carried out by Mona Mirtsch and her team[51]. The group used web mining techniques to retrieve data about a wide landscape of German firms, either Small-to-Medium Businesses (SMB) and large enterprises.

The outcomes (shown in figure 5.1) bring to light that the growing pace which was expected for the standard adoption among the last 10 years has not been respected at all. Reasoning for this insight has been identified into two main points: firstly, a wide part of the companies that have been taken into account use to take advantage of commercial partners with a certification to indirectly claim their compliance with the standard. In fact, SaaS companies and data centers are usually big enterprises that can easily afford the certification. Therefore, it is convenient for smaller customers to take advantage of such. Indeed, it is no coincidence that the study showed that SMBs are the company segment with the lowest adoption rate of the standard. The second reason, related to this last information, is that a large portion of firms can also implement their management system standard, simply avoiding to seek the certification because of the related costs.

Despite the if the research is limited to Germany, there are reasons to think that the country has been evaluated as representative of the entire continent. In fact, the research has been mainly supported by the European Commission.

In the light of the latest EU Cybersecurity Act, the authors of the article point out that, although the adoption of the standard is currently completely voluntary, it cannot be excluded that it may become mandatory in the near future.

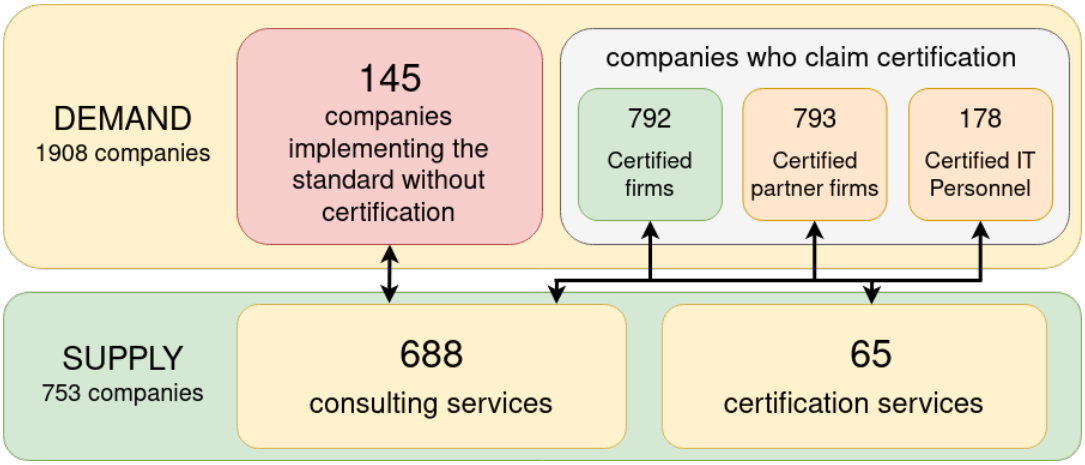

Figure 5.1: ISO 27001 adoption statistics from [51].

5.2 Design phase comparison

The fast pace of developments and the continuous changing of technologies should suggest to companies how important would be to share security responsibilities among developers, security team and operations. Based on the 2020 Snyk report[35], an improvement under this point of view has been noticed with respect to the previous years. The interviewees move from a completely unbalanced and unhealthy developers responsibility towards a more distributed burden with respect to either *software* and *infrastructure*. Despite the encouraging answers, the same report indicates also that firms are lacking in programs to increase this kind of culture across personnel. In fact, among some common practices with this culture growth aim, 47% of the population reported to have any of those in place. Therefore, still a lot can be achieved to really establish a balanced perception of security culture.

Another important step that should be done in the design phase is the selection of tools and packages that will be involved in the development. A package cannot be choose solely with respect to the technical solutions provided. Software engineers, together with security team should carefully evaluate their possibilities. Snyk report reveals an encouraging statistics over the interviewees. In fact, 65% of them

reported to check if an active community is present behind the project, 59% analyzes the frequency of commits and releases, 58% evaluate the ratings and number of downloads and the 53% even review known vulnerabilities.

Less encouraging is the figure for the adoption of threat modelling sessions. Unfortunately, only 19% of respondents use this methodology effectively.

5.3 Implementation phase comparison

As already discussed in section 3.4, software engineers may tend to prefer the development phase by underestimating the review of what they produce. Even though neglecting review phase of your own code is a very difficult task, using the correct tools may lead to increasing the improve the final result, even before pushing the commit to the repository. On the same line, restricting capabilities of a service or forcing the usage of some protocols for the sake of security may sound as a task which decrease user experience without bringing effective advantages.

5.3.1 Linters usage

As demonstrated by the research conducted by [52], the usage of *linters* seems to be strongly linked with the quality of the delivered software. In the study, the authors took into account the top 30000 starred Javascript repositories in GitHub, analyzing their configuration files to detect the usage of linters. As shown in table 5.1, there is an inverse correlation between the number of project taken into account from the ranking and the related linters usage. In fact, the more low starred projects are involved in the analysis, the more the linter adopters percentage decrease, until reaching 28,5% when the entire ranking is taken into account. Instead, 7 out of the top 10 repositories uses at least one or more of these tools. Taking into consideration all the repositories involved in the used dataset, the percentage decrease again to

24,2%. These latter data testify the poor usage of a specific kind of tool that, looking at the top of the table 5.1, developers should struggle to adopt more and more within their development environment.

Top projects	**Projects with linter**	**% of top projects**
10	7	70%
100	65	65%
300	185	61.7%
1000	535	53.5%
3000	1371	45.7%
5000	2082	41.6%
10000	3675	36.8%
20000	6306	31.5%
30000	8554	28.5%
83892	**20292**	**24.2%**

Table 5.1: Estimation of linters' usage for 30000 top starred GitHub Javascript projects, from [52].

5.3.2 HTTP Security Headers usage

HTTPS response headers are a powerful way of enforcing the usage of specific measures that drastically increase the security of a web application. Despite the remarkable usefulness, an analysis over the usage of such tool conducted by Artūrs Lavrenovs and F. Jesús Rubio Melón[53], evidenced the surprisingly low rate of adoption. Among the one million websites used as test bench for the research, the 29.1% of all HTTPS requests shown incorrect TLS configuration due to untrusted Certificate Authority (CA), self-signed certificates or expired ones. Implementation

rates are also linked to the popularity of the website itself. An exponential negative pattern in adoption rate for all the security headers has been registered following the popularity curve of the web application taken into account. Specifically, HTTP Strict Transport Security policy is used by the 17.5% of the entire web resources, while a 38% of adoption rate has been recorded considering only the first one thousand most popular websites. Only a 1.6% of the one million resources uses CSP, while 50% uses *HttpOnly* flag and 19.3% adopt the *Secure* directive. Information leakage has been registered to be more frequent among less popular or HTTP sites.

In addition, a recent study conducted by Eman Salem Alashwali[54] highlighted HTTPS inconsistencies for application and transport layers. A new type of vulnerability, called "region confusion", aims to take advantage of the different HTTPS header protections that the same request from different countries returns as a result. The paper also provides practical examples of how an attacker can exploit such inconsistencies.

5.4 Test phase comparison

According to the 2020 Snyk report[35] still the 67% of the interviewees reported to manually review and discover vulnerabilities inside their code. Still, the 48% of the population uses audit tools to analyze source code, either automatically of firing the tool manually. In fact, audits can be easily integrated into pipelines to get the most out of them together with other sort of controls.

Although the importance of CI/CD processes has been highlighted in section 2.4.2, the DZone trend report[55] presents a good understanding over the real usage of such tool. Continuous Integration is almost a widely-adopted practise, since the 89% of the teams reported to have automated pipelines to build and test the code. Following the same nomenclature enforced in 2.4.1, this percentage decrease the more we move toward production: tests are automatically performed in *STAGE*

environment by the 80%, but then in *QA* and *PROD* environments the percentage goes down with respectively 57% and 50%. That being said, more than half of the respondents (55%) indicate CI as useful to catch problems in the early stage, estimating around 30% of executed pipelines as failed because of an undetected error.

According to the 2020 Snyk report[35], the most commonly integrated security control in pipelines is the SAST, with 57% of adoption among the respondents. Apparently, SAST is the only control applied by the majority of companies since only 20% and 19% reported to also respectively use DAST and dependency scanner tools. Despite Kubernetes is the choice of 44% of companies that leverage over a container-based infrastructure, only by 28% of the respondents are using security tools to analyze their manifests.

Speaking about dependencies, their importance and the risk that those introduce into the source base of a web application seems to be not that well perceived by most of the organizations: 60% of the organizations involved in the report admit to miss the analysis of the full dependency tree of their source base, inspecting only partially the tree or omitting to check it at all.

Back to DZone report[55], Continuous Deployment follows a diminishing trend of usage nearby the production stage, likewise CI gait. In fact, the 59% reported to automatically deploy from QA to PROD, followed by 73% and 85% following the steps backwards. 77% instead, use to deploy to DEV as soon as a commit has been created. A good practice can be to apply security test cases together with smoke tests in the CD pipeline: 28% of Snyk report respondents perform such security controls in QA environment.

DZone report announce a remarkable increment of investments in CI/CD methods in the last two-three years for SMBs. Despite the positive effects, the integration of CI/CD methods within the software development process is not immediate. The

DZone report states that SMBs can achieve the complete process within six up to nine months, but Large enterprises timing is estimated between six months up to two years. Even though this shifting periods can be expensive for an organization, later outcomes (security-wise or not) are remarkable.

5.5 Deploy phase comparison

Supporting the web application once it has been deployed means also patch effectively the system as soon as new potential security holes are discovered. Therefore, the ability to detect vulnerabilities is important, even if it is even more crucial being able to solve as soon as possible. According to 2020 Verizon Data Breach Investigation Report[1], 81% of the breaches are contained within days or even less. This statistic is quite predictable given the severity that can result from an intrusion. When it comes to fixing vulnerabilities, the readiness to react decreases as there is no tangible emergency, although the risk may still be very high depending on the situation. 2020 Snyk report[35] once again provide a good overview on the case: only 1% of the flaws are fixed within a day from the detection; less then 20 days is required to handle the 34% of vulnerabilities, while 29% of those are solved in 20-70 days. The remaining 36% of vulnerabilities took more than 70 days in order to be purged. A reasonable comparison can be made with data provided by 2020 GoSecure report[50], which states that weeks or even more are needed to patch over the 52% of the vulnerabilities.

It is quite reasonable to think that the urgency of the patch strongly depends on severity of the vulnerability itself. Perhaps, this is not enough to justify so long reaction timings, which by average leaves even too much time to potential attackers to exploit the flaws.

6 Implementation and verification of the approach: a case study

According to the 2020 Verizon Data Breach Investigation Report[1], the common shift toward the usage of cloud computing and SaaS solutions hardly contribute to narrow down the difference of attack rate between large enterprises (which used to be widely more attractive) and SMB. Even if the split is still high (407 incidents for SMB against 8666 for large enterprises in the last year), Verizon reports that 28% of the overall breaches involve SMB. A rising phenomenon, it is useful to observe how security is treated from a small firm's perspective.

Having the possibility to have an internal perspective in one of the companies who decided to implement and use a microservice architecture is a great opportunity to deeply understand how the workflow is built on the architecture itself. The author of this thesis had the chance to work for Awake.ai[1], a young startup which proposes an innovative online platform addressed to businesses working in the maritime logistics field. The company is composed by less then 30 employees in which a small security team is included. After roughly one year of activity, the company managed to deliver the Most Valuable Product (MVP) of a web platform which provides to the customer useful insights and optimization proposals for commercial maritime traffic.

Likewise the previous chapter, the SDLC main structure will be used as backbone

[1]Awake.ai website: `https://www.awake.ai/`

to verify the alignment of the company with either the best practices (chapter 3) and the real scenario analysis (chapter 5).

6.1 Requirements

Since confidential information is handled by the platform, ensuring that the traffic is secure in every single step of the dataflow is a priority for the company. Therefore, it is quite important for the startup to be compliant the ISO 27000 certification. Security team fixed the certification as a milestone and is working in order improve what is missing to be fully compliant with the standard. Meanwhile, every time an architectural decision has to be taken, the security team facilitates the discussion in order to always take a decision in line with the ISO 27001 standard.

6.2 Design

While software engineers design the architecture of the software, they may need to integrate the usage of a new library. If that happens, security team is usually consulted to receive a feedback regarding the third party software and its reliability.

The company uses a cloud computing solution where several different environments are set up, in line which what described in the section 2.4.1. The development environment is used as a test bench for testing new features and patches, therefore every developer is encouraged to use that space without any fear of disrupting the service. Running each single service locally is still possible and almost every service is documented with a detailed procedure which explains the steps to take in order to run the software locally.

The infrastructure is scripted using infrastructure-as-code tools, therefore it is easy to reestablish a working state of the services in case something goes wrong. Disaster recovery guidelines are well documented as well and describe every single

step that is needed to set from scratch the entire infrastructure.

Secrets are handled using the related service offered by the SaaS provider and is well adopted among all the teams. Quite a lot of effort has been spent in the last periods before the MVP to centralize the handling of secrets. In addition, the company is also evaluating the usage of an ad-hoc third party solution which would allow to decrease dependency from the cloud computing firm, as well as offering more advanced tools such as the periodic key rotation.

Threat Modeling was not taken into account from the very beginning, but is now under process. Most important components have been already analyzed and the rest of the services are in plan. The Data Flow Diagram is used as baseline to start the discussion, while the STRIDE pattern is usually always visible during the meetings to keep in mind crucial points to think about. No special tools are used than a standard whiteboard in addition to the DFD. Usually the head of security team facilitate the meeting, where the involved developers and the security team members take part.

6.3 Implementation

During the implementation phase the developers are free to use and test tools which can improve the flow. Linters are one of those: it is a software engineer's choice to setup a linter in his environment. Therefore there are no written conventions over the code style. Nevertheless, linters are integrated into the pipelines, basically working as SAST. Therefore, at least a basic scan of the code is enforced and the commits can also be rejected in case an error is raised.

Speaking about HTTP Security Headers, security team struggled to work in cooperation with developers to enhance the overall security level of the platform before the MVP launch. At first, the usage of HTTPS over HTTP has been forced using redirects, while CORS and CSP headers, together with Cookies directives

have been configured to restrict as much as possible the attack surface. Particular attention has been dedicated also to all those headers which can leak useful insights over the system configuration.

6.4 Test

More than 50 smoke tests are usually fired before committing a deploy in production. Tests are made against the QA environment using different browsers. Together with standard functional tests, few security-related ones are included like checks against XSS attacks or similar.

No automated penetration tests or fuzzing have been configured yet. What has been developed is a dependency scanner. The tool is deployed as a container within the cluster and it is fired once per day during the night. Only the dependencies that eventually end up in the production software are tested and alerts in the internal chat are pushed in case some vulnerability has been discovered. Because of the characteristics of microservices, each container can be developed with a different programming language. Therefore, the scanner goes through all the different services and, one by one, detect the used languages and fire relative tools against the codebase.

Recently, container scanning capabilities have been implemented in the company. Thus, DevOps and security team decided to follow the same idea of the dependency scanner, creating a parallel service that can run during night as an additional service.

6.5 Deploy

Since the company is using an agile development process, deployment is usually forced at the end of each sprint (which can last 2 or 3 weeks). Such deadline is not that strict in reality since a deployment can be semi-automatically made just by

tagging a certain repository version as release candidate.

Once the development team creates a release candidate, a CD pipeline is fired and a small amount of integration test is made. The involved developers are called to notify the tester that will fire the complete set of smoke tests and, eventually, will allow software engineers to create deploy to production as well. Such workflow is not completely in line with the CI/CD philosophy since many manual steps are still needed in order to deploy a definitive version into the final product, but still the workflow is lean enough to avoid the acceptance of product owners.

Analytics over the services and the entire platform is available and generates alerts for DevOps in case strange behaviors are detected. A monitoring tool is adopted by the company, which allows to retrieve useful insights and logs from the each single service and from the cluster in general. On top of this information, different dashboards are available to every employee and are generally used to monitor performance and traffic load among the platform. Since the simplicity of usage, each developer is encouraged by the DevOps team to craft their own dashboards.

7 Conclusion

A deep analysis over the most up-to-date technologies, best practices and recommendations to ensure web applications based on microservices architecture has been conducted. Ensuring a web platform effectlively is a task that needs to be accomplished working all along the lifecycle that the software development follows. After analyzing the ideal Secure Software Development Lifecycle for a company that adopts microservice architecture, a comparison with the actual measures taken on average by companies was drawn.

The study showed a clear negative detachment between the current state of the art and the real approach adopted by the firms. Based on the multiple technical reports analyzed and in line with the expectations endorsed by the data on attacks in 2020 reported by Verizon, none of the various phases of the Software Development Lifecycle can be considered sophisticated in terms of measures taken to avoid possible cyber attacks.

A virtuous case study was analyzed to understand how much and how the size of the business can influence the implemented methodologies. Even if not in line with all the recommendations, the small startup proved to be definitely above average compared to the previous analysis, suggesting a real advantage brought by the dynamism of the team and the ability to make changes frequently and quickly, unlike large enterprises. However, it is not possible to generalise these observations to all SMBs. Therefore, a possible future study could try to consider more com-

panies of this size to see if the characteristics found are in line, consolidating the considerations made in this study.

As an additional future research option, it would be extremely useful to explore how to accelerate progress for more structured companies, where changes takes always reasonably more time. In fact, an enterprise needs to carefully evaluate all the cascade effects of a possible change may provoke, while also the application itself may require many steps to be accomplished and more policies to be respected.

Acronyms

ACL Access Control List. 38

APIs Application Programming Interfaces. 12, 13, 50, 65

AWS Amazon Web Services. 66

CA Certificate Authority. 71

CAB Change Approval Board. 27

CD Continuous Deployment. 27, 28, 73, 79

CI Continuous Integration. 25–28, 72, 73

CI/CD Continuous Integration and Deployment. 11, 24–26, 28, 47, 64, 72, 73, 79

CIS Center of Internet Security. 59

CISSP Certified Information Systems Security Professional. 61, 62

CNCF Cloud Native Computing Foundation. 17

CORS Cross-Origin Resource Sharing. 49, 50, 77

CPU Central Processing Unit. 9, 16

CSP Content Security Policy. 50, 72, 77

CSRF Cross Site Request Forgery. 50, 51

CSS Cascading Style Sheets. 50, 52

DAST Dynamic Application Security Testing. 54, 62, 63, 73

DevOps Software Development Operation. 3, 6, 19, 22, 32, 55, 56, 59, 61, 64, 65, 78, 79

DevSecOps Software Development Security Operation. 60

DFD Data Flow Diagram. 40–42, 45, 77

DNS Domain Name System. 18, 57

DOS Denial of Service. 53

HIPAA Health Insurance Portability and Accountability Act. 34

HTTP Hypertext Transfer Protocol. 49, 52, 72, 77

HTTPS Hypertext Transfer Protocol Secure. 41, 42, 51, 71, 72, 77

IAST Interactive Application Security Testing. 62, 63

IDE Integrated Development Environment. 47

IEC International Electrotechnical Commission. 34

InfoSec Information Security. 3, 56

IP Internet Protocol. 18, 49, 59

ISMS Information Security Management System. 35

ISO International Organization for Standardization. 34, 35, 68, 69, 76

K8s Kubernetes. 17–19, 58–60, 73

LFS Log-structured File System. 8

LINDDUN Linkability, Identifiability, Non-repudiation, Detectability, Disclosure of information, Unawareness, Non-compliance. 40

MVP Most Valuable Product. 75, 77

OS Operating System. 13, 14, 16, 42

OWASP Open Web Application Security Project. 45, 46, 48, 49

PaaS Platform as a Service. 8

PASTA Process for Attack Simulation and Threat Analysis. 41

PCI-DSS Payment Card Industry Data Security Standard. 34

RAID Redundant Array of Independent Disks. 8

SaaS Server as a Service. 66, 68, 75, 77

SAST Static Application Security Testing. 54, 62, 63, 73, 77

SDLC Software Development Lifecycle. 5, 7, 32, 33, 44, 45, 67, 75, 80

SMB Small-to-Medium Businesses. 68, 73–75, 80

SOC Security Operations Center. 65

SOP Same Origin Policy. 50

SSDLC Secure Software Development Lifecycle. 32, 33, 48, 80

STRIDE Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege. 40, 42–44, 77

TLS Transport Layer Security. 41, 53, 71

TM Threat Modeling. 36, 39–42, 44–46, 77

TRIM Targeted Review of Internal Models. 40, 43

TTL Time To Live. 39

UI User Interface. 9

URL Uniform Resource Locator. 51

VM virtual machine. 14

XSS Cross-site Scripting. 48, 50, 78

References

[1] Verizon, *2020 data breach investigations report*, 2020. [Online]. Available: https://enterprise.verizon.com/resources/reports/dbir/ (visited on 10/24/2020).

[2] Verizon, *2019 data breach investigations report*, 2019. [Online]. Available: https://web.archive.org/web/20190726222804/https://enterprise.verizon.com/resources/reports/dbir/ (visited on 10/24/2020).

[3] R. Hoda, N. Salleh, and J. Grundy, "The rise and evolution of agile software development", *IEEE software*, vol. 35, no. 5, pp. 58–63, 2018.

[4] K. M. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. J. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development", 2001. [Online]. Available: https://agilemanifesto.org/principles.html.

[5] CollabNet and VersionOne, *13th annual state of agile report*, 2018. [Online]. Available: https://stateofagile.com/#ufh-i-613553418-13th-annual-state-of-agile-report/7027494 (visited on 10/24/2020).

[6] CollabNet and VersionOne, *1st annual state of agile report*, 2007. [Online]. Available: https://stateofagile.com/# (visited on 10/24/2020).

[7] A. Stellman and J. Greene, *Learning agile: Understanding scrum, XP, lean, and kanban*. United States of America: O'Reilly Media, Inc., 2014.

[8] J. Sutherland and J. Sutherland, *Scrum: the art of doing twice the work in half the time*. Currency, 2014.

[9] K. Rindell, S. Hyrynsalmi, and V. Leppänen, "Busting a myth: Review of agile security engineering methods", in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, New York, USA, 2017, pp. 1–10.

[10] N. M. Mohammed, M. Niazi, M. Alshayeb, and S. Mahmood, "Exploring software security approaches in software development lifecycle: A systematic mapping study", *Computer Standards & Interfaces*, vol. 50, pp. 107–115, 2017.

[11] N. B. Ruparelia, "Software development lifecycle models", *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 3, pp. 8–13, 2010.

[12] A. Alshamrani and A. Bahattab, "A comparison between three sdlc models waterfall model, spiral model, and incremental/iterative model", *International Journal of Computer Science Issues (IJCSI)*, vol. 12, no. 1, p. 106, 2015.

[13] K. Petersen, C. Wohlin, and D. Baca, "The waterfall model in large-scale development", in *International Conference on Product-Focused Software Process Improvement*, Springer, Berlin, Heidelberg, 2009, pp. 386–400.

[14] N. Forsgren, D. Smith, J. Humble, and J. Frazelle, "2019 accelerate state of devops report", Tech. Rep., 2019. [Online]. Available: `http://cloud.google.com/devops/state-of-devops/`.

[15] B. Haskins, J. Stecklein, B. Dick, G. Moroney, R. Lovell, and J. Dabney, "Error cost escalation through the project life cycle", in *INCOSE International Symposium*, 1, vol. 14, Hoboken, New Jersey: Wiley Online Library, 2004, pp. 1723–1737.

[16] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang, "Serverless network file systems", in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, New York, USA, 1995, pp. 109–126.

[17] Red Hat, *Virtualization: What is a hypervisor?* [Online]. Available: `https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor` (visited on 10/24/2020).

[18] J. Thönes, "Microservices", *IEEE software*, vol. 32, no. 1, pp. 116–116, 2015.

[19] S. Newman, *Building microservices: designing fine-grained systems*. United States of America: O'Reilly Media, Inc., 2015.

[20] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment", *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[21] D. Liu and L. Zhao, "The research and implementation of cloud computing platform based on docker", in *2014 11th International Computer Conference on Wavelet Actiev Media Technology and Information Processing (ICCWAMTIP)*, IEEE, Chengdu, China, 2014, pp. 475–478.

[22] Docker, *Docker overview*. [Online]. Available: `https://docs.docker.com/get-started/overview/` (visited on 10/24/2020).

[23] T. Combe, A. Martin, and R. Di Pietro, "To docker or not to docker: A security perspective", *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, 2016.

[24] L. E. Hecht, *CoreOS, Red Hat and Kubernetes Competition*. [Online]. Available: `https://thenewstack.io/coreos-red-hat-kubernetes-competition/` (visited on 10/24/2020).

[25] Kubernetes, *Kubernetes concepts*. [Online]. Available: `https://kubernetes.io/docs/concepts/` (visited on 10/24/2020).

[26] Google Container Tools, *"Distroless" Docker Images*. [Online]. Available: `https://github.com/GoogleContainerTools/distroless` (visited on 10/24/2020).

[27] Docker, *Use multi-stage builds*. [Online]. Available: `https://docs.docker.com/develop/develop-images/multistage-build/` (visited on 10/24/2020).

[28] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. United States of America: Pearson Education, 2010.

[29] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices", *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[30] K. Beck and E. Gamma, *Extreme Programming Explained: Embrace Change*. Massachusetts: Addison-Wesley Professional, 2000.

[31] L. Chen, "Continuous delivery: Huge benefits, but challenges too", *IEEE Software*, vol. 32, no. 2, pp. 50–54, 2015.

[32] J. Bird, *DevOpsSec: Delivering Secure Software Through Continuous Delivery*. United States of America: O'Reilly Media, 2016.

[33] Unity Technologies, *Unity - Secure Software Development Lifecycle SSDLC*. [Online]. Available: `https://github.com/UnityTech/unity-ssdlc/blob/master/Overview.md` (visited on 10/24/2020).

[34] G. Disterer, "ISO/IEC 27000, 27001 and 27002 for information security management", *Journal of Information Security*, vol. 4, no. 2, pp. 92–100, 2013.

[35] Snyk, *State of open source security report (2020)*, 2020. [Online]. Available: `https://info.snyk.io/sooss-report-2020` (visited on 10/24/2020).

[36] K. Bernsmed and M. G. Jaatun, "Threat modelling and agile software development: Identified practice in four norwegian organisations", in *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, IEEE, Oxford, United Kingdom, 2019, pp. 1–8.

[37] N. Shevchenko, T. A. Chick, P. O'Riordan, T. P. Scanlon, and C. Woody, "Threat modeling: A summary of available methods", Carnegie Mellon University Software Engineering Institute Pittsburgh United, Tech. Rep., 2018.

[38] K. Wuyts and W. Joosen, "LINDDUN privacy threat modeling: A tutorial", *CW Reports*, 2015.

[39] L. Sion, K. Yskout, D. Van Landuyt, and W. Joosen, "Solution-aware data flow diagrams for security threat modeling", in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, New York, USA, 2018, pp. 1425–1432.

[40] OWASP Foundation, *OWASP top 10 web application security risks*. [Online]. Available: `https://owasp.org/www-project-top-ten/` (visited on 10/24/2020).

[41] Mozilla, *HTTP observatory scoring methodology*. [Online]. Available: `https://github.com/mozilla/http-observatory/blob/master/httpobs/docs/scoring.md` (visited on 10/24/2020).

[42] OWASP Foundation, *OWASP secure headers project*. [Online]. Available: `https://wiki.owasp.org/index.php/OWASP_Secure_Headers_Project#ect` (visited on 10/24/2020).

[43] B. P. Miller, *Fuzz testing of application reliability*. [Online]. Available: `http://pages.cs.wisc.edu/~bart/fuzz/` (visited on 10/24/2020).

[44] S. S. Malladi and H. C. Subramanian, "Bug bounty programs for cybersecurity: Practices, issues, and recommendations", *IEEE Software*, vol. 37, no. 1, pp. 31–39, 2019.

[45] Center of Internet Security, *CIS kubernetes benchmark version 1.6.0*, 2020. [Online]. Available: https://www.cisecurity.org/benchmark/kubernetes/ (visited on 10/24/2020).

[46] Snyk, *The untold tale of helm chart security*, 2019. [Online]. Available: https://snyk.io/wp-content/uploads/helm-report.pdf (visited on 10/24/2020).

[47] PureSec Ltd., *Ten Most Critical Risks for Serverless Applications*. [Online]. Available: https://github.com/puresec/sas-top-10 (visited on 10/24/2020).

[48] E. Conrad, S. Misenar, and J. Feldman, *Eleventh Hour CISSP®: Study Guide*. United States of America: Syngress, 2016.

[49] R. Mullins, B. Nargi, and A. Fouse, "Understanding and enabling tactical situational awareness in a security operations center", in *Advances in Human Factors in Cybersecurity*, Cham, Switzerland: Springer International Publishing, 2020, pp. 75–82.

[50] GoSecure, *Cybersecurity perceptions versus reality*, 2020. [Online]. Available: https://landing.gosecure.net/W-Report-Cybersecurity-Perceptions-Versus-Reality-Landing.html (visited on 10/24/2020).

[51] M. Mirtsch, J. Kinne, and K. Blind, "Exploring the adoption of the international information security management system standard iso/iec 27001: A web mining-based analysis", *IEEE Transactions on Engineering Management*, pp. 1–14, 2020.

[52] K. F. Tómasdóttir, M. Aniche, and A. Van Deursen, "The adoption of javascript linters in practice: A case study on eslint", *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 863–891, 2020.

[53] A. Lavrenovs and F. J. R. Melón, "HTTP security headers analysis of top one million websites", in *2018 10th International Conference on Cyber Conflict (CyCon)*, IEEE, Tallinn, Estonia, 2018, pp. 345–370.

[54] E. S. Alashwali, P. Szalachowski, and A. Martin, "Exploring https security inconsistencies: A cross-regional perspective.", *IACR Cryptology ePrint Archive*, vol. 2020, p. 79, 2020.

[55] DZone, *The state of CI/CD*. [Online]. Available: `https://dzone.com/trend reports/the-state-of-cicd` (visited on 10/24/2020).