POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Smart Contract and DevSecOps

Supervisor:

Prof. Danilo BAZZANELLA

Candidate

Diego ZANFARDINO

Company Supervisor:

Emiliano ORRÙ

October 2023

To my family

Summary

Recent years have brought significant changes in software development practices. The demand for continuous development has given rise to DevOps methodologies, which emphasise iterative and ongoing practices over sequential activities. Therefore, automation for development and operation practices have become widely adopted in enterprise scenarios, enabling individuals with different roles and responsibility to perform actions on the pipeline that regulate software development. With the automation of security measures and controls (DevSecOps), these actions and roles have gained increased importance and liability.

The automation of security processes can lead to increased risk, both for the delivered product and the DevOps infrastructure as a whole. Unauthorised access, for instance, may result in unapproved changes reaching the production environment. Therefore, the purpose of this thesis work was to apply *security by design* principle through the use of modern technologies in order to mitigate these threats. To further improve system security and accountability for each entity's action, smart contracts were identified as a suitable solution. To validate the feasibility of the proposed approach, a Proof of Concept was built in association with a cybersecurity team of Security Reply.

The presented system involves designing a permissioned blockchain network which can communicate with an enterprise-grade DevSecOps platform, secure digital assets and record executed operations. To enable the smart contract to receive off-chain information, a custom oracle was implemented that would relay received data to the network. Given the potential target of such a solution, and to provide an additional layer of security, an Hardware Security Module has been integrated in the proposed architecture to act as a Root of Trust and securely store sensitive information.

Acknowledgements

Firstly, I would like to express my gratitude to Professor Danilo Bazzanella for giving me the opportunity to work on this thesis.
I extend my thanks to Emiliano and his team at Security Reply S.r.l for their cordial welcome and the provision of the necessary technological resources for development. I am thankful to Fabio, Andrea and Dario for their constant support and feedback during this project, they have proved to be excellent mentors to aspire to.
Finally, I would like to express my gratitude to those with whom I shared this extraordinary journey at the Politecnico, without which this would not have been possible.

Table of Contents

List of Tables

List of Figures

Acronyms

API
Application Programming Interface

BCCSP
Blockchain Crypto Service Provider

CA
Certificate Authority

CI/CD
Continuous Integration and Continuous Deployment/Delivery

CVSS
Common Vulnerability Scoring System

DAST
Dynamic Application Security Testing

DLL
Dynamic Link Library

DLT
Distributed Ledger Technology

DSL
Domain Specific Language

DSO
DevSecOps

HF

Hyperledger Fabric

HSM

Hardware Security Module

IPFS

InterPlanetary File System

MSP

Membership Services Provider

PBFT

Practical Byzantine Fault Tolerance

PoC

Proof of Concept

PoW

Proof-of-Work

PKCS

Public Key Cryptography Standard

RoT

Root of Trust

RNG

Random Number Generator

SAST

Static Application Security Testing

SDL

Software Development Life cycle

SDK

Software Development Kit

SO

Shared Object

SoD

Segregation of Duties

SPoF

Single Point of Failure

TRNG

True Random Number Generator

VSCC

Validation System ChainCode

XSS

Cross Site Scripting

Chapter 1

Introduction

Software development methodologies have profoundly changed over the past decades. They consist of a framework to plan, build and control the process of application development. Traditionally it was a sequential process, where security teams would assess and solve vulnerabilities before the software was deployed. The need of continuous development has led to the emergence of DevOps methodologies, where development (Dev) and operations (Ops) practices are iteratively and continuously performed. Together with continuous and frequent development needs, strict requirements for what concerns the security of the developed applications became integral to the development process, since the speed of development and release must not undermine security. DevSecOps has been conceived to merge together in a unified SDL (Software Development Lifecycle) Development, Security and Operation. The core principle that enabled the transition from DevOps to DSO (DevSecOps) is the "shift-left" approach, in which security controls and requirements are introduced as early as possible in the classical DevOps pipeline. This principle ensures that security is prioritized from the initial stages of development, reducing the number of vulnerabilities and their impact.

Different groups of people, with different roles and responsibilities, have to access and modify information on the CI/CD pipeline in order to push development forward. For each action there could be associated risks and liabilities. Currently existing DSO tools, give the user the possibility to implement custom security policies. Those policies may require a minimum number of approvals to effectively accept incoming changes or a minimum number of allowed vulnerabilities in new code. Since each of these action is performed via one's personal account, it is not exempt from the dangers of loss of credentials and account

compromise.

A new possible implementation was therefore sought to address these concerns. The innovative solution would provide the necessary liabilities and immutability to each of the performed actions, thus improving also the overall security of the stored information. Among all the most recent technologies, smart contracts were identified as a suitable solutions for our needs. In fact they provide a mechanism to design and securely store self-executing contract that can guarantee, upon correct design and implementation, the aforementioned security measures. The definition of the immutable code stored on the blockchain will administer all the interactions of the different parties involved with the information stored on the blockchain.

DSO organizational patterns are typically used with large code bases that are subject to continuous change, usually in an enterprise scenario. This consideration is taken into account when choosing a specific blockchain or framework for PoC development. In this particular use case the above requirements become even more critical. Any action or change performed in the DSO pipeline can result in major losses both from a financial perspective but most importantly in terms of users privacy and security.

1.1 Objectives

The main objective of this thesis was to investigate and develop a Proof of Concept that could improve the existing DSO model and interaction. This work was carried out at Security Reply S.r.l. consulting company, whose focus is on cybersecurity and personal data protection. The company also provided the necessary technological means useful to build the PoC such as the physical hardware and the HSM. More in details here are the objectives of this thesis work:

- Research for state of the art and already existing solution for smart-contract integration with DSO architecture;
- Identification of the best platform and framework that better suits the requirements;
- Realization of a PoC;
- Study of main concept related to HSM (Hardware Secure Module) and

PKCS#11 standard and implementation in the existing PoC in order to increase the security of the overall solution.

1.2 Outline

The remaining sections of the document are structured as follows:

- Chapter 2 - *Background:* this section aims to provide a basic background knowledge of the main topic of this thesis work;
- Chapter 3 - *State of the art:* in this section the current state-of-the art for smart contract and DSO integration is analyzed;
- Chapter 4 - *Solution design:* in this chapter, following the workflow that led to the PoC, the main design choices are analyzed;
- Chapter 5 - *PoC implementation:* this chapter describes the actual implementation of the PoC;
- Chapter 6 - *Conclusion and future development:* this final section summarises the result obtained and provides some analysis for future development.

Chapter 2

Background

This thesis was characterised by a high level of multidisciplinarity and modern technologies and concepts. Therefore, this section aims to provide a solid background to fully understand all the topics covered, such as blockchain and DevSecOps. Both concepts have become very successful in recent years: DSO because of its advantages compared to previous software development methodologies, while blockchain because of its technological novelty, which could have many different applications to improve the security of already existing systems.

2.1 DevSecOps

DSO has been developed in recent years as an evolution of DevOps practises. Key concept for DevSecOps development are traceability, verifiability and visibility. More specifically DevSecOps describes an organization's cultural and technical practices, aligning them in such a way to enable the organization to reduce the gaps between a software developer team, a security team, and an operations team. [1] The thing that most characterize this organizational pattern is the constant interaction both between members of different DSO team and also with product stakeholders.

Together with security, the other two main aspects of DSO are Continuous Integration (CI) and Continuous Delivery/Deployment (CD) which enable the coupling of development and operation practises. Those concept are translated into practice through the building of a pipeline composed of several jobs and stages, according to the needs, that automatically react to some performed action.

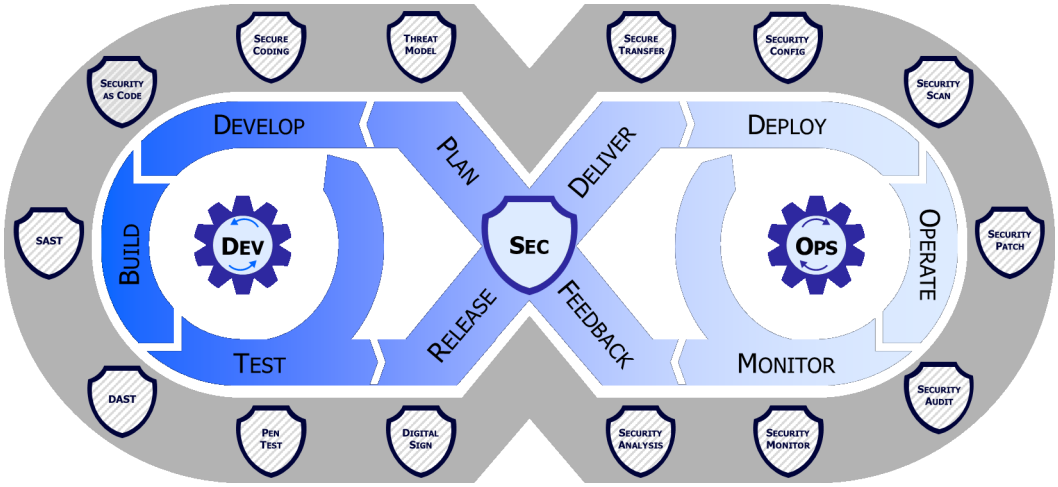

Figure 2.1: DevSecOps Lifecycle Phases and Philosophies [1]

2.1.1 CI/CD

This practise is used to automate frequent integration of changes made by different developers in a central repository. It is typically divided into stages, each of them having their input needed and artifact produced during the stage. More in details some of them could be the following:

- During the initial **planning** stage, general requirements to understand scope and objectives are discussed between DSO team and stakeholders. During this phase deadlines and security needs are defined and agreed;

- During the **development** phase, developers make changes in their local copy of the code implementing the planned features. Typically in this phase no artifact is produced by the pipeline but there's the possibility to run tests locally before pushing in the main repository;

- **Build** phase should be automated. It is possible to specify policies in order to trigger the build phase. It is also possible to build Docker container using specific Dockerfile in order to carry out test at later date. Once the build is complete, it is possible to run some static analysis on the entire application, in order to find common vulnerabilities like *XSS* (Cross Site Scripting) or *injections*. More specifically through SAST (Static Application Security Testing) tools is possible to perform white box testing, therefore heavily reducing the review code effort [2]. SAST tools are considered the most important security within the whole SDL[3];

- During the **test** phase, after the build one, it is possible to perform also different type of tests. For security concerns DAST (Dynamic Application Security Testing) analysis on the running application is one of the most used and effective. Those black box tools allow to find a different spectre of vulnerabilities with respect to SAST analysis. In both cases, artifacts containing the found vulnerabilities are produced during the pipeline stages. If containers are used, some additional security checks on those can be performed;
- Once the test are successfully passed, in order to release, deliver and deploy the application it is possible to require:
  - a minimum number of approval
  - a minimum number of found vulnerabilities

This last phase marks the application's transition to **operation** stage.

By defining **approval rules** for merge request, it is possible to set the minimum number of required approvals before work can merge in the project. It is possible to define as *approvers* users with specific permissions or user belonging to specific groups. It is also possible to select the level of granularity of those approval rules: project, merge request or instance level. After a merge request receive the required number of approvals, the corresponding is automatically merged unless merge conflict of failure of following stages of the pipeline are encountered.

It is also possible to set a minimum number of vulnerabilities in order to make the commit proceed through the pipeline: these controls are called **security gates**. These gates can be designed in order to be triggered if a specific number and type of vulnerabilities is found, developers and stakeholders are typically informed and the current pipeline running can be automatically stopped according to specific design. For example, if one or more critical vulnerability are found, the deployment stage is not reached.

2.1.2 DSO framework vulnerabilities

Several tools and utilities need to be used in DSO scenario, typically a complete and existing framework is preferred, rather than performing and implementing all the needed actions individually and then integrate them all together in a

single and continuous flow.

As it is possible to see from [4] more than 800 vulnerabilities have been found in the last five years in Gitlab, that is one of the most popular tools used to perform DSO activities in enterprises.

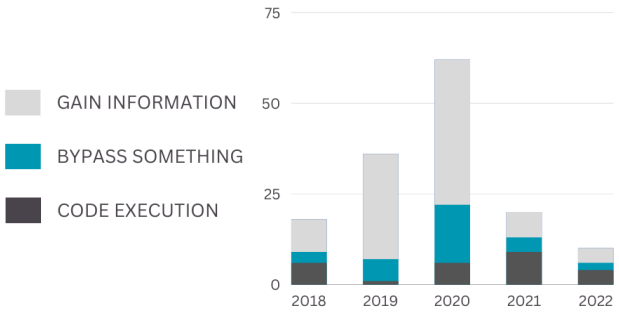

Figure 2.2: Gitlab vulnerabilities in recent years

These are the most relevant vulnerabilities for the scope of this thesis project. As a matter of fact they give unauthorized users the possibility to gain information or perform malicious actions. As an example *CVE-2019-2428*, discovered in early 2020 with a CVSS (Common Vulnerability Scoring System) score of 7.5, in which users could bypass the mandatory external authentication provider sign-in restrictions by sending a specially crafted request. In this way some of the security controls mentioned above can be maliciously bypassed.

2.2 Blockchain

Blockchains are tamper evident and tamper resistant digital ledgers implemented in a distributed fashion and usually without a central authority. At their basic level, they enable a community of users to record transactions in a shared ledger within that community, such that under normal operation of the blockchain network no transaction can be changed once published. Blockchains are composed of cryptographically signed transactions that are grouped into blocks. Each block is cryptographically linked to the previous one (making it

tamper evident) after validation and undergoing a consensus decision. As new blocks are added, older blocks become more difficult to modify (creating tamper resistance). New blocks are replicated across copies of the ledger within the network, and any conflicts are resolved automatically using established rules. [5].

This technology has seen widespread success in recent years, mainly due to its adoption in the development of cryptocurrencies, like Blockchain end Ethereum. Apart for its use in financial sector as the basic element of decentralized and electronic cash, the underlying technology has the potential to be broadly applicable in different domains and scenarios. Key concept of blockchain technology are:

- Security and integrity: it is chryptographically secure, ensuring that data contained in the ledger are tamper proof;
- Distribution: the ledger is shared among different nodes of the network;
- Auditability: all the changes in the blockchain network remain stored in the ledger.

The above-mentioned characteristics closely align with the DSO principles outlined in section 2.1

2.2.1 Operation

The main component of blockchain technology are cryptographic hash functions. They are used to compute a unique digest for nearly any input size data. Cryptographic hash functions have the following important characteristics [5]:

- *Preimage resistance*: they are *one-way* function. It is impossible to compute the correct initial value given some output;
- *Second preimage* resistance: given a specific input it is computationally infeasible to find a second input which produces the same output. That is, given x, is impossible to find y such that $hash(x) = hash(y)$, unless exhaustive search on the entire input space is used;
- *Collision resistance*: one cannot find two inputs that hash to the same output. It is computationally unfeasible to find x and y such that $hash(x) = hash(y)$.

In a blockchain network a transaction is an interaction between different parties. In an enterprise scenario that could also be used as a way of recording

activities occurring on physical or digital systems. Each block in a blockchain can contain zero or more transactions and the ledger is a secure, append-only collection of these transactions.

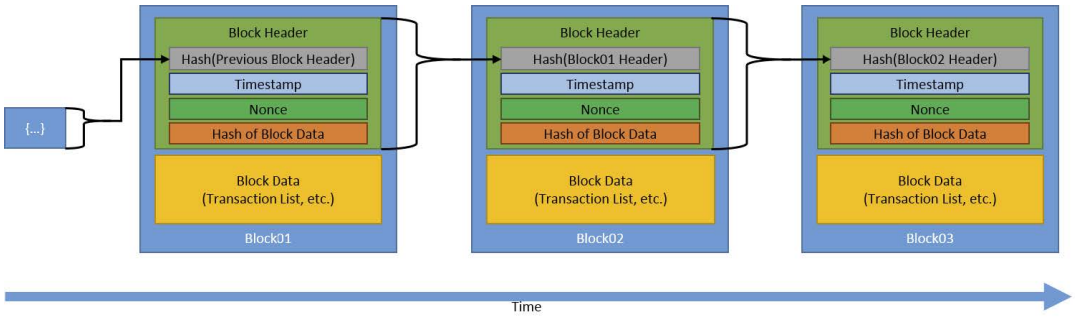

Figure 2.3: Blockchain structure [5]

In figure 2.3 it possible to see how the chain of blocks is composed. Together with data section, that contains the information on the transactions, each block can contain other essential material in order to prevent chain tampering. Each specific blockchain implementation can define its own fields and block structure, but many of them include some field with the following function:

- Block Header:
  - The hash of the previous block header;
  - A timestamp of when the transaction is submitted;
  - A nonce;
  - The hash of the data block.
- Block Data

2.2.2 Permission

Blockchain can be categorized by their permission model, that is how user can publish blocks on the network. It can be:

- **Permissionless** (also called public): decentralized ledger platforms open to anyone publishing blocks, that do not require any permission from any authority. Anyone that can publish blocks have the visibility to read all the published transactions as well. These types of blockchains are typically free and open source. Due to their public nature, they are the most subject

to attack that try to control the system. In order to prevent this, consensus mechanism have been established in order to publish a new block;

- **Permissioned** (also called private): these type of network require some authority, either centralized or decentralized, to assign an identity and authorize user in order to publish new block. Since only authorized users maintain the network, it is possible to restrict the read access to authenticated individuals only. The main advantage of this permissioned networks is that, being private, they do not require a consensus algorithm as complex as the permissionless one. The members sustaining the blockchain need to have an already established level of trust in order to receive an identity and submit transactions. For those reasons permissioned blockchain are typically faster and less computationally expensive with respect to permissionless ones. Since this type of network do not perform, by choice, any type of anonymisation or pseuodnymisation, can be used to achieve a shared business process with the assurance of uniquely identifying the malicious user in case of fraud or misbehaviour.

2.2.3 Consensus mechanism

The integrity of the system is guaranteed by its protocol operation and can be verified at any time by any user by checking the entire chain correctness. In order to determine which user will publish the next transaction consensus mechanisms are used to ensure integrity and consistency among all peers of the network.

The first block in the chain is called *genesis-block* and set the initial state for the new blockchain network. Every new transaction is then added after it, once consensus among the publishing peers is reached. These algorithms can differ widely when designed for a permissioned or permissionless environment.

In private networks, among the most commonly used consensus algorithms [6] we can find the following ones:

- PoW (Proof-of-Work): this protocol was first used in Bitcoin. Using this consensus protocol, each node calculates an hash value of the block header. The block header contains a nonce that is modified by the miners and consensus is reached when the calculated value is equal to or less than a certain given one. When the correct value is reached, the node would broadcast the block to other nodes who would verify the correctness of the

hash value. If the block is validated by other blocks, it can be added to any peer chain [7];

- PBFT (Practical Byzantine Fault Tolerance): this protocol is a replication algorithm to tolerate byzantine faults [8]. This consensus protocol could handle up to 1/3 malicious byzantine replicas. Two main roles exists: *clients* are the external entities that submits transactions to the PBFT networks, while *replicas* are the nodes that make up the core consensus group. Figure 2.4 shows the algorithm flow that starts whenever a client C makes a transaction request. The subsequent stages of the process can be divided into five phases:

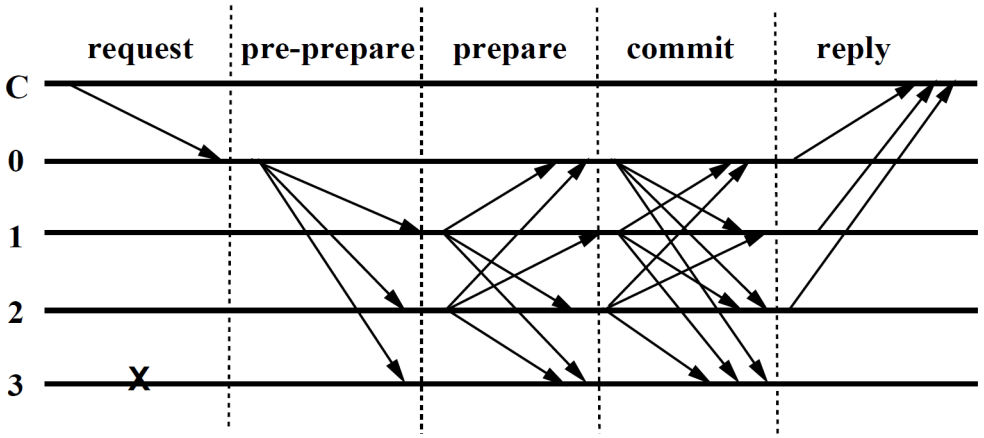

Figure 2.4: PBFT algorithm flow [6]

  - *request*: in each round a *primary* replica would be selected and it is responsible for ordering the transactions;
  - *pre-prepare*: the primary replica assigns a sequence number n and broadcasts the transaction proposal to all other replicas in the same group;
  - *prepare*: in this phase each replica that received pre-prepare message verifies proposal's validity, checking against pre-prepare view, sequence number and digest. If the received message is accepted, then each replica sends a *prepare* message, otherwise it does nothing. The prepare state is said to be true when the replica has collected at least $2f$ *prepare*, where f is the number of faulty replicas the system can tolerate;

- *commit*: when prepared becomes true the replica multicasts a *commit* message. Replicas accept commits messages and insert them in their log if they're verified. A replica is *committed* if and only if prepared check is true for all i in some set of $f + 1$ non faulty replicas. A replica is *committed-local* if and only if prepared is true and it has accepted $2f + 1$ commits (possibly including its own) from different replicas;
- *reply*: each replica executes the operation requested by client C when *committed-local* is true and the state reflects the sequential execution of all requests with lower sequence numbers. After executing the requested operation, *reply* message is sent back to client.

The pre-prepare and prepare phases are used to totally order requests sent in the same view even when the primary, which proposes the ordering of requests, is faulty. A view in PBFT refers to a specific point in time during which the algorithm operates. Prepare and commit phases instead, are necessary to ensure that requests that are successfully committed are fully ordered across views.
The algorithm provides both *safety* and *liveness* assuming no more than $\lfloor \frac{n-1}{3} \rfloor$ replicas are faulty, where n is the total number of replicas in the group. In figure 2.4 it is possible to see that replica number 3 fails and do not respond to pre-prepare, prepare and commit messages, nonetheless consensus is reached.

2.2.4 Smart Contract

Smart contracts can be defined as the computer protocols that digitally facilitate, verify, and enforce the contracts made between two or more parties on the blockchain [9]. Smart contracts are typically deployed on and secured by blockchain, and therefore inherit the characteristics of the underlying technology:

- The code, being recorded and stored on the blockchain, is tamper-resistant; therefore can be used as a trusted third party;
- The execution of smart contracts is enforced among trustless individual nodes without centralized control and coordination;
- These contracts can include digital assets or cryptocurrency and transfer them whenever predefined conditions are met.

Depending on the implementation, smart contracts can be written in specific programming languages, such as *Solidity* and *Vyper*, or using already existing

non-specific programming languages, such as Go, JavaScript and Rust.

They are deployed on the blockchain like normal transactions, but can be invoked by peers, who send a transaction proposal along with some additional information if the contract requires it. All nodes that execute the smart contract must derive the same result from the execution, and the results are recorded on the chain. Therefore the smart contract must be deterministic: given an input the same output will always be produced based on that input. All the nodes executing the smart contract at the end of the proposed transaction must agree on the final state. The distinction between permissioned and permissionless is also extended in this context as well:

- In **permissionless** blockchain network, such as Ethereum, the user issuing a transaction will typically have to pay some fee, usually referred as *gas*, to cover the cost of code execution. A limit on how much execution time can be consumed by a call to a smart contract is set. This not only allows the publisher to be rewarded for executing the smart contract but also prevents malicious users from deploying and accessing smart contracts resulting in a denial of service;
- In **permissioned** blockchain network, such as Hyperledger Fabric, it is not necessarily required to pay some fees to propose transactions. These networks are built and designed between known peers. Smart contracts can be used to ensure that the agreement between the parties is honoured, but there is no need to prevent fraudulent activity in this scenario, as any fraudulent action can be traced back to a specific known peer.

Smart contracts, however, need to overcome different challenges due to their nature. First of all, the fact that they're immutable once published in the chain, leads to several restriction on their dynamism. This can limit their applicability and gives a lot of responsibility to smart contract developers: in case the published smart contract is wrong or contains some vulnerabilities it is no longer possible to modify it, so malicious actors can exploit it.
Another major issue is the interaction between data contained in blockchain network (*on-chain data*) and data coming from external sources (*off-chain data*). This problem will be discussed in the following section.

2.2.5 Oracles

In this domain oracles act as a gateway between on-chain and real-world data. Blockchain is designed to ensure that all the information it contains is accurate

as each transaction is validated before being added to the chain and secured once stored. Incorporating information coming from outside the chain can be a threat to this assumption. In case the information reported by the Oracle is wrong the whole network may operate the smart contract on wrong data, hence leading to a loss of trust and compromise in integrity. Oracles do not produce data themselves, they simply collect data coming from other sources and report those data to the network.

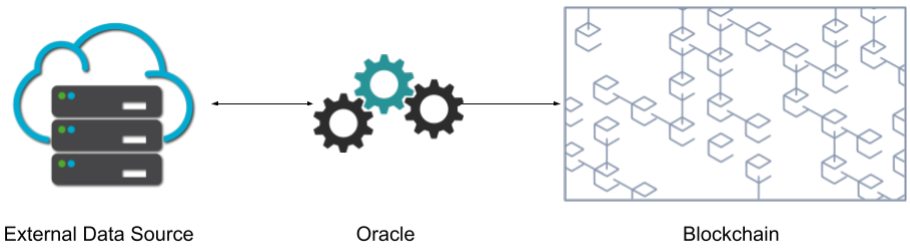

Figure 2.5: Oracle function for blockchain network

These are the main issues with blockchain oracles [10]:

- *Data reliability*: there's the need to ensure the correctness and accuracy of the provided information. They tend to change quickly and, given the distributed environment, it is difficult to get all the peers to agree on the correct value to use in transactions. They may even operate on non-deterministic data;
- *Single Point of Failure*: they introduce centralization in a decentralized system, being the oracle the centralized authority that provides information. This also lead to SPoF in case the oracle gets compromised or stops working, even if the collected data are correct;
- *Violating trustless environment promise*: blockchain was designed to be implemented in trustless environments. If an oracle is introduced in the network there's the need to trust both the data source and the oracle itself;
- *Latency*: the desire to maintain the decentralized structure of the network even for oracle, results in a high level of communication overhead. The oracle may need to aggregate data from different sources and eventually perform computations. All these factors may lead to increased latency for the whole network.

2.3 HSM (Hardware Security Module)

Cryptographic devices normally perform two types of operations [11]:

- **Storage** of the cryptographic objects such as asymmetric keys, symmetric keys and *X.509* certificates;
- Performing cryptographic **operations** such as asymmetric key pair generation, symmetric key generation, hashing, encryption, decryption and signing.

Several cryptographic devices exists with different capabilities and capable of performing different cryptographic operations.

Hardware security modules are dedicated hardware devices that physically and logically secure cryptographic keys and cryptographic processing [12]. They are used to protect sensitive data such as encryption keys and digital signatures from unauthorized access and tampering. They perform generation, distribution, rotation, storage, termination, and archival functions over cryptographic information.

Unlike cryptographic smart cards, which are used to store and manage personal keys and sensitive data, HSMs are typically used in organizations, where many objects needs to be stored and used. Together with this difference, HSMs are capable of performing complex cryptographic operations very efficiently having high-performance dedicated hardware.
Because of their size and targeted marked they're typically equipped with a TRNG (True Random Number Generator), that generates random numbers based on an high-entropy source, such as unpredictable physical processes or phenomena [13]. The use of a TRNG, as opposed to a simple RNG (Random Number Generator), ensures that generated keys and cryptographic data are not predictable or susceptible to cryptographic attacks based on the predictability of pseudo-random number generation.

HSMs are available in a variety of forms: as standalone network attached appliance, as hardware cards that plug into existing network-attached systems or as portable USB-connected HSMs that connect to a client system. The network HSM manages cryptographic storage and operations in a centralized, high assurance appliance providing a root of trust for sensitive cryptographic data transactions [12].

RoT (Root of Trust) is the fundamental concept that enables the creation of a trustworthy foundation within a system. It is the component upon which the whole system relies to ensure integrity, confidentiality and authenticity. It is typically an hardware device that is completely trusted by all the members of the system either because is hardwired or cryptographically resistant to tampering and unauthorized access. HSMs can act as a RoT in an environment where asymmetric cryptography is used, thus providing key storage and operations.

The following security measures are the key to HSM reliability:

- **Layered Encryption**: no object is left unencrypted on the device. Objects are encrypted by multiple layers and fully decrypted in volatile memory when needed [14]:

  - One general storage key (GSK) is used to encrypt general storage objects possibly needed by different roles;
  - User storage keys (USK) are used to protect the contents of the partition accessed by that role;
  - Master tamper keys (MTK) are used to strongly encrypt each object generated and stored within the HSM;
  - Key encryption key (KEK) is used to further encrypt every used key to ensure that they are never showed in plaintext.

  Each HSM or partition belong to a security domain, also called cloning domain. Object can be copied between partitions that share the same security domain. In this way it is not possible for attackers to copy stored material to an unauthorized device not sharing cloning domain with the one where cryptographic object are stored.

- **Tamper Protection**: it is typically ensured by strong cryptographic algorithm, intrusion-resistant and tamper-evident hardware. In the event that a security breach is detected, HSMs have the ability to lock themselves until a user with a specific role solves it or the HSM is reset.

HSM allows its physical space to be divided into logical partitions, each with independent data, access controls and policies. This allows multiple application sharing one single HSM without fear of compromise from other partition residing on it.

2.3.1 PKCS#11

As explained in the previous paragraph, there are several types of cryptographic devices, possibly from different manufacturers. In order to enable a common interface to communicate with these devices, RSA Security has developed the PKCS (Public Key Cryptography Standards) family. In particular, PKCS#11 standard specifies an API (Application Programming Interface) for devices that hold cryptographic information and perform cryptographic functions [15].

This standard, also known as cryptoki, isolates an application from the details of the cryptographic device. In this way the application doesn't need to change in order to interface to a different type of device, making the application highly portable. To enable this technology independence and resource sharing capabilities, an object based approach is followed.
Each object consists of a set of attributes, each with a given value. In figure 2.6 it is possible to see an high level view of Cryptoki objects and their attributes. They are generally defined in one of four classes [16]:

- Data objects: objects defined by an application;
- Certificate objects: digital certificates such as X.509;
- Key objects: public or private keys;
- Vendor defined objects.

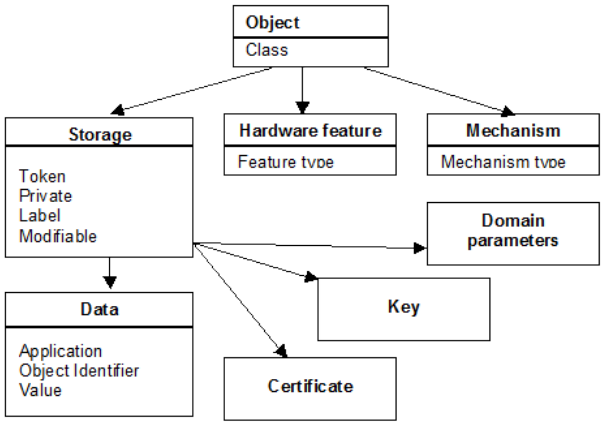

Figure 2.6: PKCS#11 object attribute hierarchy [15]

The API model is distributed as C header files, which are part of PKCS#11 standard specification [11]. It is then up to all suppliers to implement these header files and distribute as DLL (Dynamic Link Library) for Windows or as SO (Shared Object) for Linux operating system. Developed application can then load these DLL or SO files to access the cryptographic device. Here in the following PKCS#11 specific terminology:

- **Slot**: uniquely identifies an HSM logical space. One physical HSM can be composed of multiple logical partitions;
- **Token**: it is the physical device where applications store object and perform cryptographic operations. When a slot is initialized in HSM then the token is present in the slot;
- **Session**: is the logical connection between a slot and the corresponding token. Once session is established it is possible to use the session object together with the token to store information or perform operations. This relationship, better visualized in figure 2.7, allows for stronger isolation and security when applications connect to HSMs.

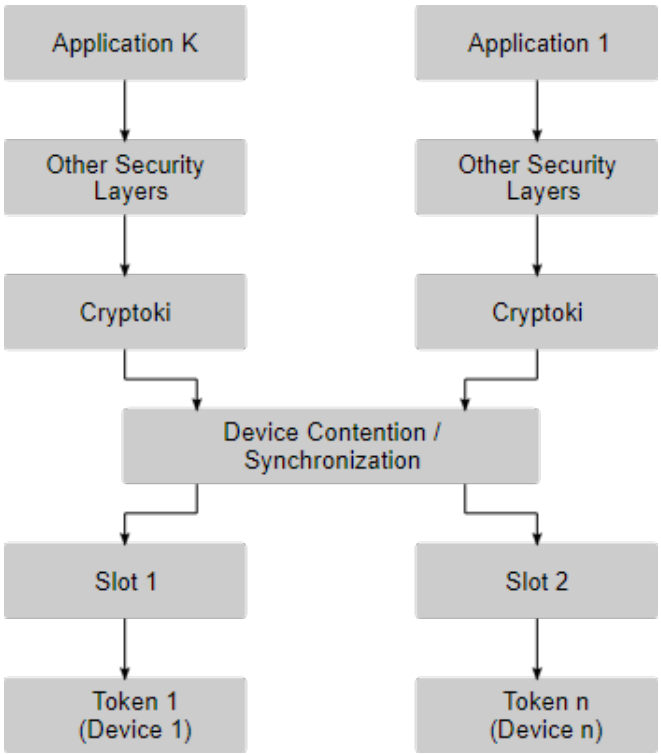

Figure 2.7: PKCS#11 interaction model [16]

- **User**: cryptographic devices contains both private and public objects. To access private objects, such as private keys, users must be authenticated from the device. In order to adhere to PKCS#11 standard at least two roles must be present: the security officer (SO) and standard users. Luna network HSMs define an enhanced version of PKCS#11 role hierarchy described above. Each role grants permission to perform specific action at a different level of granularity, offering great freedom for personalisation and providing great security when properly configured. The following tables summarize the possible roles and their functions [17]:

Appliance-level role	**Description**
Admin	– Performs all administrative and configuration tasks on the appliance. – Creates custom users and roles with access to specific sets of commands.
Operator	– Can perform administrative tasks on the appliance. – Cannot execute any commands that affect other roles on the appliance
Monitor	– Executes commands that present information about the appliance and HSM, cannot affect the state or contents of the appliance or HSM.
Audit	– Manages HSM audit logging

Table 2.1: Thales Luna HSM Appliance-level role

HSM-level role	Description
Security Officer (SO)	– Initializes the HSM, creating SO credential – Creates/deletes application partition and configure policies – Must have admin-level access to the appliance in order to perform all actions
Auditor	– Manages HSM audit logging – Must have audit-level access to the appliance in order to perform all actions

Table 2.2: Thales Luna HSM-level role

Partition-level role	Description
Partition Security Officer (PSO)	– Initializes the partition, creating PO credential, setting cloning domain and configuring partition policies. – Initializes Crypto Officer (CO) role and can reset its credential
Crypto Officer (CO)	– Creates and modifies cryptographic objects on the partition. Can perform cryptographic functions via user applications – Initializes Crypto User role and can reset its credential
Crypto User (CU)	– Performs cryptographic functions via user application but can create public object only

Table 2.3: Thales Luna Partition-level role

Chapter 3

State of the art

During the initial phase of this thesis work, existing solution that shared requirements and solutions with what was described previously. Various proposals have been presented in recent years, which will be analyzed in this chapter in order to gather the current state of the art for blockchain and DSO integration via smart contracts.

3.1 Transparent continuous delivery

In this article [18] presented by Kimberly Connors *et al.* blockchain technology is seen as a possible solution to enable a transparent SoD (Segregation of Duties) compliance, in order to increase delivery efficiency and agility. SoD is an important security requirement that aims to prevent errors and fraudulent activities through separation of critical tasks and responsibilities between different individuals or roles.
Key concepts of SoD that are common to those of this dissertation proposal, together with error detection and compliance, are the following:

- **Fraud prevention**: division of responsibilities results in fewer problems with unauthorized access and control over processes that could be exploited to perform underhanded activities. This concept is strongly reflected in blockchain networks, where the entity giving consent to add a transaction is different from the one proposing it;
- **Accountability**: responsibilities are shared between different individuals. This increases transparency. The use of smart contracts and blockchain is intended to improve the accountability of the systems in which they are applied;

- **Conflict of interest**: decision making from processes and decisions are made from entities with different roles.

A specific DevOps model, called Bi-modal DevOps, is used for the analysis. This methodology combines both agile and waterfall to implement changes in the systems. This enables a strict gate before the product reaches the production stage. Here are some of the delivery controls that have been identified and can be audited in a DevOps model: management of delivery artifacts, traceability of code changes, authorization for builds and promotion of code into delivery and production environment, role-based access control and logging.

In the proposed solution, summarized in figure 3.1, a blockchain based code repository would offer a decentralized solution to immutability and auditability.

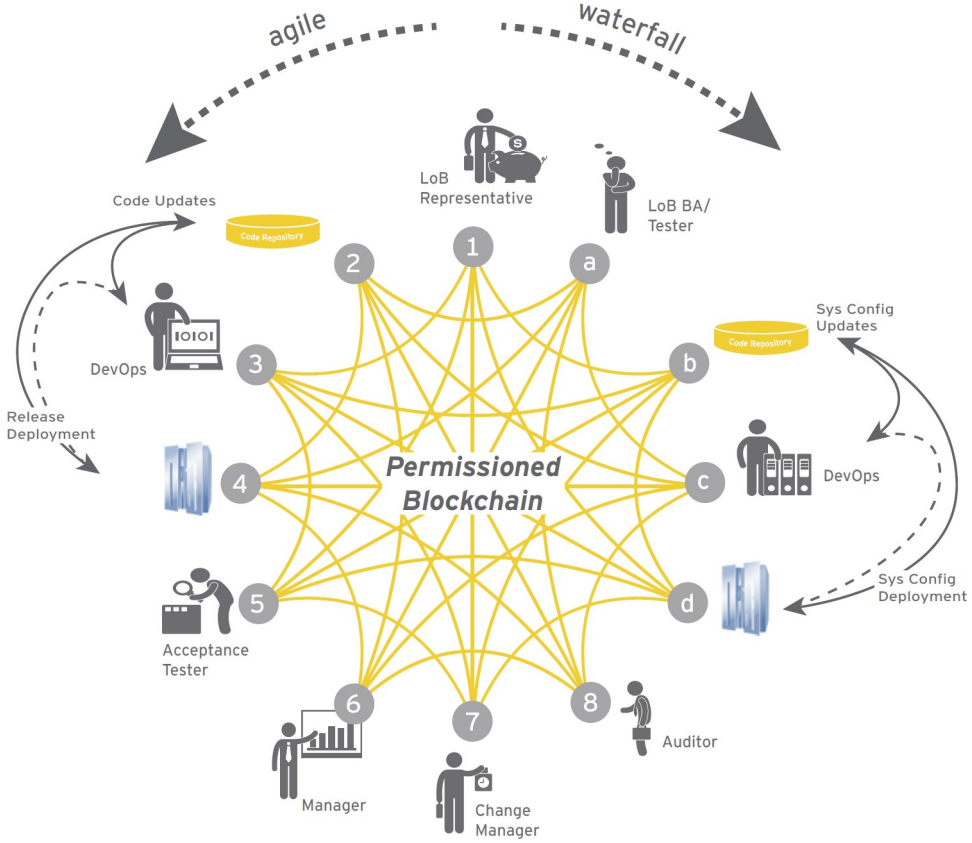

Figure 3.1: Bimodal DevOps using blockchain [18]

This suggested solution, also called Enterprise DevOps Blockchain (EDOB), has been designed in such a way that:

- all activities are automatically recorded as transactions;
- all network participants run nodes in EDOB, implemented as permissioned blockchain;
- transactions can be independently validated and processed.

The proposed framework, following counterclockwise direction of figure 3.1, is able to perform the following actions:

1. The specific Line of Business (LoB) problem is captured by initializing the blockchain and setting requirements and needs;
2. Code repositories are integrated in EDOB and activities are treated as transactions in the network;
3. DevOps activities are recorded as well. For example new code developed, test results and code promotion to production environment can be some of the relevant information;
4. Deployment to production should be a fully automated process and is also recorded in EDOB, along with any access to production activities and other automated vulnerability code scanning results when in production environment;
5. Smart contract can be used to trigger acceptance testing, both manual or automatic. Test scripts and results can be recorded as transactions in EDOB;
6. Management and responsible executives have complete transparency over the project and all its milestones;
7. In case an approval from change management is needed, it can be notified and the outcome can be recorded as well in the blockchain;
8. At any point, internal or authorized external auditor can review the process and access every transaction performed in every step of the activities.

EDOB solution offers a trusted data source for transactions recorded on chain, improved security and increased efficiency and agility of delivery process. Moreover the distributed nature of the project makes it fault-tolerant and very

scalable, relying on peer-to-peer network where each node is managed by the respective network participant.
Although the main purpose of the analysed solution is to improve and facilitate compliance with modern and more stringent SoD policies, it is certainly possible to draw useful information and practices from this specific use case, which will be taken into account in the solution design.

3.2 Blockchain based framework for software development using DevOps

In this document [19], presented at the 2021 International Conference on Nascent Technologies in Engineering, a different approach is described with respect to the previously cited solution. The solution proposed by Bankar and Shah aims to support DevOps processes giving all stakeholders access to all the documents on a single decentralised network.

The overall architecture, shown in figure 3.2, is composed of two main components: a permissioned blockchain network and an IPFS (InterPlanetary File System) that acts as distributed database. A client SDK (Software Development Kit) allows for peers in the network, such as developers, tester and managers, to submit transaction requests.

IPFS is a modular suite of protocols for organising and transmitting data, based on the principles of content addressing and peer-to-peer networking. In such a network, each peer has the same capabilities and can initiate a communication session. In addition, unlike traditional system, each file is addressed by its hash value rather than by location or path [20]. In the mentioned solution hashes of different directories are saved in a JSON file, in this way access to those file is faster.

In continuous development, an initial phase in which client and manager interact in order to outline all the details of the project. These interactions, which take place through the client interface, are all recorded and shared across the distributed system. Once everything is set up, a git repository is created in the IPFS database. Any authorized contributor can perform CRUD (Create-Read-Update-Delete) operations on the project. The blockchain is used to authenticate and authorize the collaborator to access the IPFS. After a successful transaction on the IPFS a new hash of the project is generated and

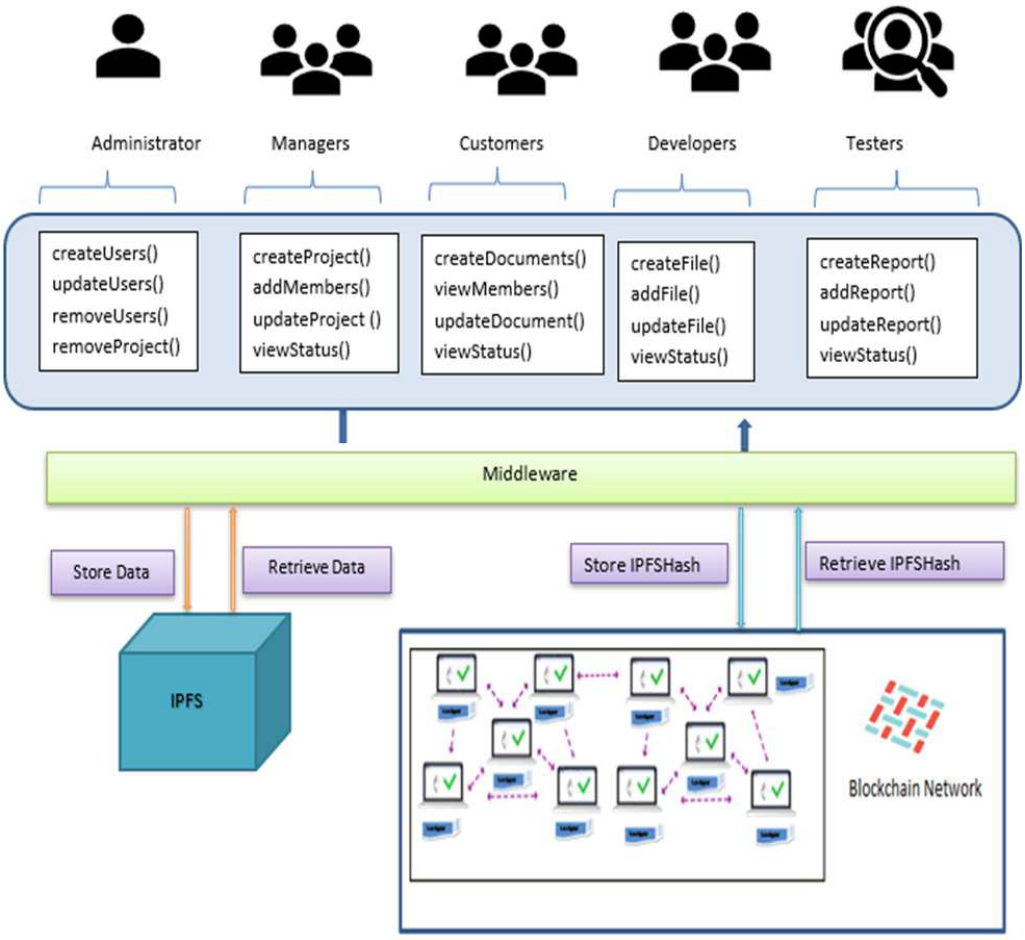

Figure 3.2: Architecture for DevOps with Blockchain

stored in the blockchain's state database. All the SDL related operations are performed on the IPFS server. Administrators have the ability to interface with the middelware in order to create new users and provide identity materials in order to enable strong authentication.

The proposed framework bring the following advantages:

- All document history is traced and origin of changes is easily recovered;
- All stakeholders can be brought on a single platform;

- All documents are stored in a tamper-proof manner thanks to blockchain technology;
- All latest update are easily available to all stakeholders at the same time.

Principles and practices previously proposed, developed for DevOps methodologies, are certainly useful and applicable when DevSecOps practices take place. These benefits will be further analyzed in following chapters and have been considered in the development of this thesis project proposal design.

Chapter 4

Solution design

This thesis work focuses on improving security when different groups of people with different roles and responsibilities need to access and modify information in a DevSecOps pipeline in order to push development forward.

In order to demonstrate the feasibility of the proposed approach a Proof of Concept has been developed. As there's currently no integrated system for combining blockchain technology with DSO, it was necessary to develop a customized solution, which required careful consideration of design choices. The following chapter explains the rationale behind each decision and outlines the benefits in terms of improved security, reliability, efficiency and accountability.

4.1 DevSecOps platform

One of the first decisions to be made is which DevSecOps platform to use. A number of choices exists today, and each offers some common and unique features, with different ways of interacting.

The aim of the following sections is to provide a summary of the most frequently used tools capabilities, so that it is possible to compare them and choose the one that best suits the needs of this project. The comparison will be between Jenkins [21], GitHub Actions [22] and GitLab [23].

4.1.1 Jenkins

Jenkins is a very flexible and extensible open source automation server. There are several plugins for source code management, build and deploy, testing and more. This variety allows Jenkins to integrate with a wide range of tools and

services. Jobs can be configured through its web interface, using different type of projects, build steps and triggers.

Pipelines are written using a DSL (Domain Specific Language) based on Groovy. Among all the solutions the are presented, this is the more customisable: users can create custom plugins if needed. The scalability of the system depends on the installation configuration. It is possible to set up a master-slave architecture to distribute workloads and handle parallel builds.

4.1.2 GitHub Actions

This CI/CD platform integrates with GitHub repositories and has the ability to automate and make software development more efficient. Steps, jobs and triggers for the CI/CD pipeline are stored inside *.github/workflows* in *YAML* files together with the repository. It is said to be event-driven, in fact, each of the action specified inside the configuration files is triggered by specific events, such as code pushes, pull requests, comments and many more. It supports matrix build, enabling to run jobs for different configurations in parallel.

A very collaborative community exists that develop and share custom actions inside the marketplace. Those actions can be reused by developers if needed.

4.1.3 GitLab

GitLab CI/CD is tightly integrated with the GitLab platform, providing a single interface for source code management, issue tracking and pipeline management. Pipelines are deployed in *gitlab-ci.yml* file, written in *YAML* markup language and versioned along with the code. GitLab CI/CD has native support for Docker containers, that can be used to build, test and deploy applications. With the added benefit of parallel and distributed build support, this can scale efficiently at both individual project and group level.

It provides *Auto DevOps* feature with the purpose of automating common pipelines based on the repository content. It offers some integration with other tools and services but most of the feature are implemented inside this framework. The Enterprise Edition (EE), depending on the subscription tier, supports most of features that enable DevSecOps development and a shift-left mindset. Some of these tools are:

- **Security testing**: it is possible to use both built in and custom scanners

in order to perform SAST, DAST, secret scanning, dependency scanning and many others;

- **Vulnerability management**: security teams can manage vulnerabilities directly from pipelines, third party tools or on-demand scans.

Some common features of the last two proposed frameworks are the following:

- They provide options for securely storing sensitive data such as API keys and credentials using secrets;
- Workflows, pipelines and configurations updates can be secured using access control mechanisms. These are linked to each platform's access controls, defining role and responsibility for each entity;
- They allow users to use self-hosted runners, which can provide additional security control by running pipelines in a self-managed infrastructure.

After an evaluation of the proposed DSO platform, GitLab was chosen for its comprehensive set of tools, on-demand technical support and high level of customisation. Jenkins would have been an unnecessary burden for this project due to the complexity of its setup and was therefore discarded. The solution was also found in this instrument because the proposed work is aimed at an enterprise environment and this DSO platform is one of the most widely used in companies worldwide.

In addition, GitLab provides the ability to easily set up and run a complete instance locally, giving the opportunity to experiment with different configurations on a local machine. Because a local installation is used for the scope of this project, runners and container registries have to be set up manually in order to run pipelines.

4.2 Blockchain

When developing a smart contract, choosing the right blockchain platform is a crucial decision that can significantly impact the effectiveness and security of the contract. The choice of blockchain platform depends on various factors, such as requirements, technical considerations and ecosystem support.

One of the critical factors to consider when choosing a blockchain platform is the type of blockchain. Permissioned blockchains, such as Hyperledger Fabric

[24] and Corda [25], are designed for private networks, and access is restricted to a selected group of participants. In contrast, permissionless blockchain such as Ethereum [26], enables anyone to participate in the network and validate transactions.

Another important consideration is the scalability and performance of the blockchain platform. Smart contracts can involve multiple parties and complex logic, making it essential to choose a platform that can handle the transaction volume and execute the contract efficiently.

Moreover, the cost of using the blockchain platform should also be considered. The cost of deploying and maintaining a smart contract can vary significantly depending on the platform's features and services. Ethereum, for example, charges a fee for every transaction on the network, while Hyperledger Fabric do not.

Below, there's a concise comparison between the most commonly used platform [27] for smart contract development:

	Hyperledger Fabric	**Ethereum**
Network type	Permissioned	Permissionless
Consensus mechanism	customisable, PBFT based	PoW (v 1.0), PoS (v 2.0)
Smart contract language	GO, JavaScript, Java	Solidity (DSL)
Scalability	efficient consensus mechanism	scalability challenges
Typical use case	Enterprise solutions	DApps [1], NFTs [2]

Table 4.1: Popular smart contract platform comparison

For the reasons that follow, which will be more comprehensively discussed in subsequent chapters, Hyperledger Fabric was selected to develop the smart

[2]DApps stands for Decentralized Application and is a type of software application that is designed to operate on a decentralized and distributed blockchain network.

[2]Non-fungible tokens

contract:

- **Scalability**: Fabric's architecture supports high scalability, enabling efficient handling of large transaction volumes and diverse workloads, making it highly scalable compared to other platforms as also reported in [28];

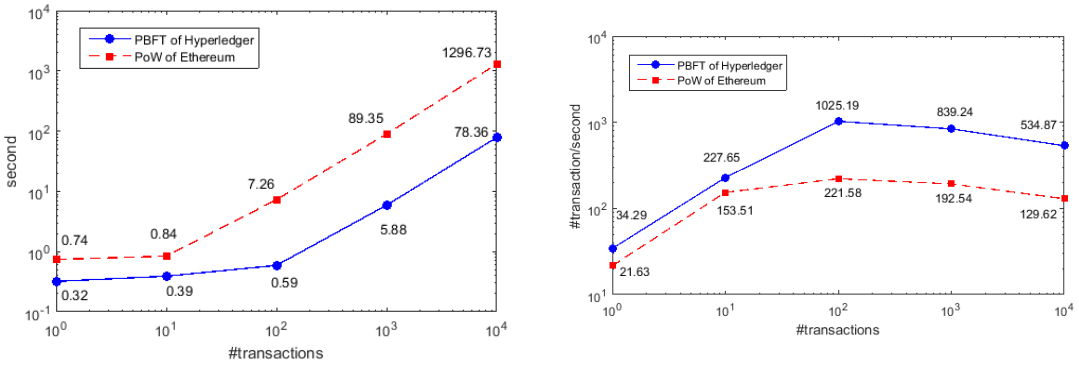

Figure 4.1: Average latency of Ethereum and Hyperledger with varying number of transactions (left); Average throughput of Ethereum and Hyperledger with varying number of transactions (right) [28]

- **Permissioned network**: Fabric is designed for permissioned blockchain networks, where all participants are explicitly known, making it suitable for enterprises use cases requiring privacy and access control.
  The specific scenario presented in this work requires to have all these characteristics: a permissionless network, where every peer could join and make transaction is discarded;

- **Modular architecture**: Fabric's modular architecture facilitates the customization of the consensus mechanism, membership services, and numerous other components. This flexibility enables customization to suit various requirements;

- **Smart contract flexibility**: Fabric's smart contracts, also known as *"chaincode"* can be written in multiple programming languages, including Go, Java and JavaScript;

- **Membership**: Fabric's membership services enable fine-grained access control and identity management. Participants must be authorized to access the network, which covers the accountability that this project requires;

- **Consensus validation**: its endorsement and consensus model separates transaction endorsement from validation, thereby improving efficiency and decreasing redundancy. It can also leverage consensus protocol that does not require a native cryptocurrency.

4.2.1 Hyperledger Fabric

According to its documentation [29] Hyperledger Fabric (HF) is an open-source enterprise-grade permissioned distributed ledger technology (DLT) platform, designed for use in enterprise contexts, that delivers some key differentiating capabilities over other popular distributed ledger or blockchain platforms. Some of these peculiar characteristics, developed together with the Linux Foundation, are the ones introduced at the end of the previous chapter and are the main needs that have driven the development of this new framework.

The majority of the existing smart-contract capable blockchain platforms follow an **order-execute** architecture in which:

- the consensus protocol is needed to validate transactions and propagate them to all peer nodes;

- each peer then executes the transaction sequentially.

These type of blockchain must be deterministic in order to reach the required consensus and a DSL, such as Solidity, is needed to address non-determinism at a different level.
Fabric introduces a new approach that is called **execute-order-validate** that addresses the order-execute model's resiliency, flexibility, scalability, performance and confidentiality. It does so by partitioning the transaction flow into three stages, better detailed in subsequent sections:

- *Executes* a transaction and check its correctness, hence endorsing it;

- *Orders* transactions via a consensus protocol;

- *Validates* transaction against an application-specific endorsement policy before committing to the ledger.

The first phase eliminates non-determinism, in this way DSL are no longer necessary and it is possible to use standard programming languages.

4.2.1.1 Network structure

As described in the official documentation [30], HF's network is the infrastructure that provides ledger and smart contract services to applications. Its distributed architecture provides an high level of transparency and accountability. In most enterprise scenarios, multiple organisations or parties can come together to form a channel where transactions are invoked through chaincodes and where permissions are determined by a set of policies are agreed upon when the channel is initially configured.

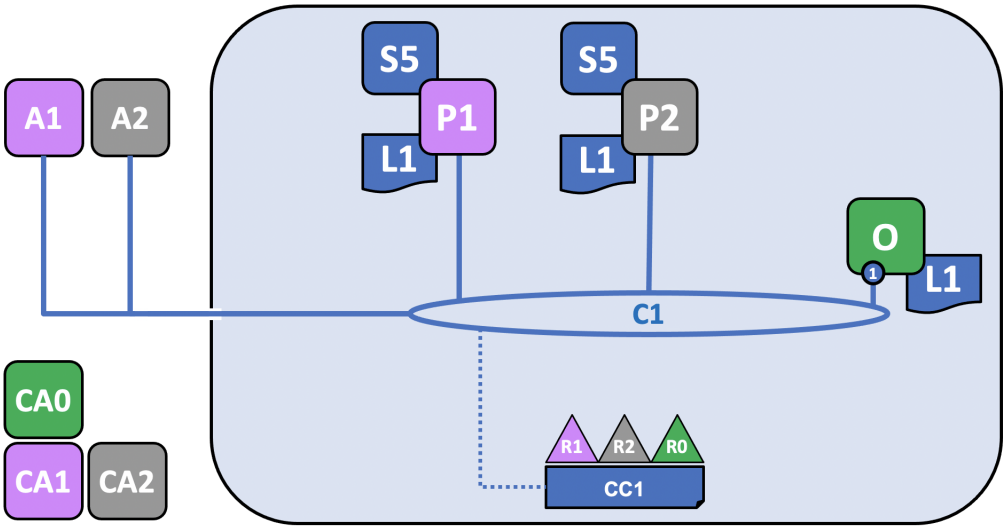

Figure 4.2: Hyperledger Fabric network [30]

In figure 4.2, it is possible to see an example of a complete network that will be referenced when explaining the infrastructure. The starting point for building a new network is to define the **organisations**, also called *members*, that will join the blockchain network. In the picture three different organizations are present: R0, R1 and R2. The **channel configuration**, CC1, has been agreed by organizations R0, R1 and R2 and is contained in the **configuration block**. This block contains a record of the organisations that can join components and interact on the channel, as well as policies that govern transactions, and is built using *configtxgen*. This is a command-line tool that has been developed to create and inspect channel configuration related artifacts, using the information contained in the *configtx.yaml* file.

Each organisation must be created by a Certificate Authority (CA) associated with each organisation, CA0, CA1 and CA2 respectively. Each CA has the

ability to register and enroll participants for each organisation, including the identities of each member's administrator. Their role is to distribute X.509 certificates that are used within the network to

- **Identify** components as belonging to an organization;
- **Sign** transactions, meaning that an organization endorses the transaction result.

The organisation's CA issues the required certificate, while the MSP (Membership Service Provider) maps this certificate to the member organisation. The channel configuration can assign rights and permissions to each organisation by defining custom policies. Issued certificates are used in the transaction generation and validation process, more specifically by the client application when a transaction proposal is issued and by the smart contract when there's a need to sign the transaction response.

Other important actors in the network are the **peers**, such as P1 and P2, respectively belonging to organisation R1 and R2. They both store the digital ledger (L1) and a copy of the chaincode (S5). Organisations can submit transactions either by using a peer to connect to a channel or by using an application. A chaincode can be considered physically hosted on a peer but logically hosted on a channel. A peer can be part of as many channels as needed, depending on its configuration.

For each channel an **ordering service** needs to be present. It collects endorsed transactions from applications and peers, orders them into transaction blocks and then distributes them to each peer node in the channel. Each committing peer records the transactions and updates the local copy of the ledger. In the figure 4.2 there is only one orderer, labelled as O and belonging to organisation R0. In a production environment it is recommended to deploy at least three orderer nodes for each channel.
It is important to note that the ordering service only includes the blockchain portion of a ledger and not the state database nor the chaincode, as they do not propose transactions.

The final component depicted in figure 4.2 is the **client application**, which can be used to invoke transaction on a chaincode, via the Fabric Gateway service. The client applications, like both peers and orderers, possess an identity associated with an organisation: A1 belongs to R1, while A2 belongs to R2.

The client application builds a gRPC [3] connection to the gateway, managing the transaction proposal and endorsement on the application's behalf.

4.2.1.2 Membership Service Provider

Since we are in a permissioned environment peers, orderers and client applications are identified by their digital identity encapsulated in an X.509 certificate. Access to resources and information within the blockchain network is enabled based on this identity. For an identity to be verifiable, it must come from a trusted authority, namely the MSP [31]. In order for a member to transact on a Fabric network it is needed to:

- Have an identity that is trusted by an organization, according to the MSP configuration;
- Check that the organization MSP is part of the channel on which the transaction is proposed;
- Ensure that the MSP is included in the policy definition of the network.

The function of the MSP is to assign each network participant a specific role according to its digital identity. That occurs in two domain inside the network, depending on the scope for which they define the role:

- **Local MSP**: it specifies permissions for a node, such as peers and orderers. Each node must have a local MSP defined so as to permit authentication of member messages outside the context of a channel and to define permissions over a specific node, such as for organization administrators;
- **Channel MSP**: it outlines administrative and participatory rights at a channel level. It establishes the authorities at a channel level, specifying the relationships between the identities of channel members (local MSP) and the implementation of channel-level policies. If greater detail is needed within an organisation, it is possible to further logically structure the organisation in different organizational units (OUs).

When issuing X.509 certificates, the OU field can specify the LoB to which the identity belongs. Employing OUs with this capacity, enable the utilization of these fields in policy definition to restrict access or in smart contracts for

[3]Google Remote Procedure Call: it is an open-source framework developed by Google that facilitates communication between distributed systems

attribute-based access control. Here is shown the certificate for an administrator belonging to an organization called "org1". The administrator's available actions are defined by the policies that regulate system resources.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            7a:8d:41:11:d5:93:85:7f:16:3c:13:0d:59:a0:d5:5a
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: C = IT, ST = Turin, L = Turin,
        O = org1.example.com, CN = ca.org1.example.com
        Validity
            Not Before: Aug 23 09:15:00 2023 GMT
            Not After : Aug 20 09:15:00 2033 GMT
        Subject: C = IT, ST = Turin, L = Turin,
        OU = admin, CN = Admin@org1.example.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:da:33:71:8b:e5:ef:6e:96:3c:33:08:24:cf:4b:
                    a2:ee:97:72:f6:62:25:b7:e2:af:3e:b8:8c:29:61:
                    02:2f:31:a0:ed:ce:28:6b:82:19:b0:02:f6:16:5b:
                    dd:10:75:05:16:95:d6:a7:b2:be:24:07:ed:ca:f9:
                    58:4a:53:a8:1a
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            ...
```

Only one local MSP exists, both physically and logically, in each node. In contrast, channel MSP are accessible to all nodes and are logically defined once in the channel configuration. Nevertheless, a channel MSP is instantiated on the file system of each node in the channel and kept synchronized via consensus. A local MSP folder is structured as follows:

- ***config.yaml***: defines accepted roles and enables OUs;
- **cacert**: this directory contains a list of self-signed X.509 certificates of the root CA trusted by this organization's MSP. This folder identifies the CA

that all other certificates must originate from to be recognised as members of the appropriate organisation;

- **keystore**: this folder is mandatory for a local MSP and contains the private key of the node. Access to this folder should be restricted to the identities that possess administrative responsibility over the peer. If an HSM is used, this folder will be empty as the private key is generated and stored internally within the secure module;
- **signcert**:this directory holds the node's certificate issued by the CA. This certificate is used to verify node's signature over transactions;
- **tlscacerts**: it contains a list of self-signed X.509 certificates for the root CA that is trusted by this organisation for secure communication between nodes via TLS.

A channel MSP includes this additional information:

- **Revoked Certificates**: if X.509 based identities are used this folder contains the list of revoked certificates in the pair {SKI (Subject Key Identifier) : AKI (Authority Access Identifier)}. This act as a normal CA's CRL plus revoking membership from an organization.

4.2.1.3 Policies

In the context of permissioned blockchain like Hyperledger Fabric policies allow members to decide which organizations can access or update the network. They specify the list of organisation that have access to a given resource and how many organisation need to agree on a proposal to update resources [32]. The policies, described in *configtx.yaml*, outline the approval necessary for changes proposed to a specific resource, which can be either an individual's explicit sign off (Signature type policies) or that of a group (ImplicitMeta type policies). This last type of policy is beneficial when group members are not fixed over time, as it does not require a particular member to sign off. There are three main section for defining policies:

- **Member organisations**: defines organisations that will be members of the channel. It is possible to specify policies for readers, writers, admins and endorsement procedures;
- **Application and roles**: these are the policies regarding important features of application channels, such as who may query the ledger, invoke a chaincode or update channel configurations;

- **Chaincode lifecycle**: this is a crucial aspect for the network's security. The policies describes the approval process when chaincode requires endorsement of organizations members and subsequent committing to the channel.

4.2.1.4 Ledger

In HF the ledger consists of:

- **World state**: it is a database that stores current values. Ledger states are represented as {key;value} pairs and simplify access to the latest value, rather than requiring traversal of the entire transaction chain to obtain the correct value. Information stored in the world state can be created, updated and deleted. Together with the key and value, the version of each record is also stored, to keep track of the changes;
- **Blockchain**: the transaction log that record all the changes occurred to reach the current world state. Unlike the world state, once data is recorded on the blockchain, it becomes immutable. It can be used at any moment to reconstruct the current world state.

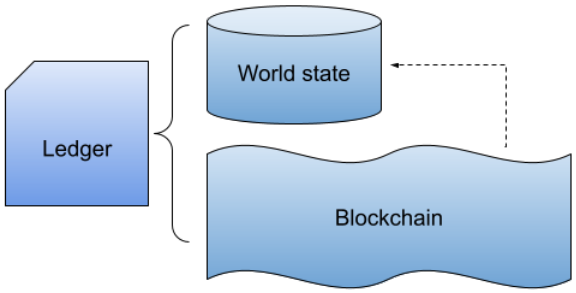

Figure 4.3: Hyperledger Fabric Ledger

The Block Data described in section 2.2.1 for this specific implementation may contain several transactions. Each of these will have the following content, the usefulness of which will be clearer once the consensus process has been explained:

- **Header**: capture relevant metadata about the transaction (i.e. name and version of the chaincode);
- **Signature**: cryptographic signature performed by the client application to ensure integrity of transactions;

- **Proposal**: when invoking chaincode it is necessary to provide some input parameters in order to update the current world state;
- **Response**: captures the before and after values of the world state as a Read Write set (RW-set);
- **Endorsement**: the list of signed transaction responses from the required organisations according to the endorsement policy.

4.2.1.5 Chaincode Lifecycle

Hyperledger Fabric smart contracts, known as chaincodes, are programs that run within a segregated Docker environment that is also isolated from the endorsing peer process. Chaincode lifecycle is the process whereby organisations agree on the parameters that define a chaincode, such as the version and endorsement policies. Four steps occurs during chaincode installation in a channel:

1. **Packaging the chaincode**: chaincode needs to be packaged in a tar file (*.tar.gz* extension) in order to be installed on peers. This compressed folder contains compressed chaincode files and another file, *metadata.json*, containing information on chaincode language, path and label;
2. **Installing the chaincode**: peer administrator is responsible for installing the chaincode on each peer participating in transaction execution and endorsement. After installation, each peer will compile the chaincode and return the respective chaincode package identifier upon success. As multiple chaincodes can be installed on a single peer, this identifier is formed by the package label and the hash of the package itself;
3. **Chaincode approval for organization**: at this stage channel members may give their approval for a chaincode definition, signifying their agreement on the parameters of the chaincode, that are common for each organisation on the channel and are the following:
    - **Name**;
    - **Version**;
    - **Package ID**;
    - **Sequence**: in the event of chaincode upgrades, this integer value is incremented by 1. The starting value is set at 1. This value ensures that all organizations remain synchronized with regard to approving and committing chaincode definition;

- **Endorsement Policy**: represent the necessary organisations needed to execute and validate the transaction output. By default, the endorsement policy requires a transaction to be endorsed by a majority of the organisations of the channel;
- **Initialization**: an *Init* function that is used to initialize the chaincode needs to be specified. This is helpful in establishing an initial state prior to the invocation of chaincode transactions by applications and peers.

Each channel member wishing to use the chaincode must submit its approval to the ordering service, which then distributes it to all peers. Once approved, it is stored in a collection available to all peers in an organisation, so that only one chaincode approval is required for each organisation.

4. **Chaincode definition commit**: when the required number of channel members have approved the chaincode definition, according to the specified policy, an organisation can commit the definition to the channel.
   The proposed commit transaction is sent to the peers of the channel members, who query the chaincode definition approved for their organisation and endorse the definition if their organisation has approved it. An administrator submits the transaction to the ordering service, which then commits the chaincode definition to the channel.

4.2.1.6 Transaction flow

As already mentioned in section 4.2.1, most blockchain systems follow an order-execute architecture, where the network first orders transaction using the consensus protocol and then executes them sequentially in all peers in the same order. Instead, Fabric's execute-order-validate paradigm involves the client, peers and orderers in the following transaction flow, also displayed in figure 4.4 [33]:

1. **Execution phase**: it begins when a client signs and sends a transaction proposal to one or more endorsers for execution. These endorsers are those specified in the endorsement policy of the chaincode. Each proposal contains:
   - The identity of the submitting client, according to MSP;
   - The transaction payload and parameters;
   - The called chaincode identifier;

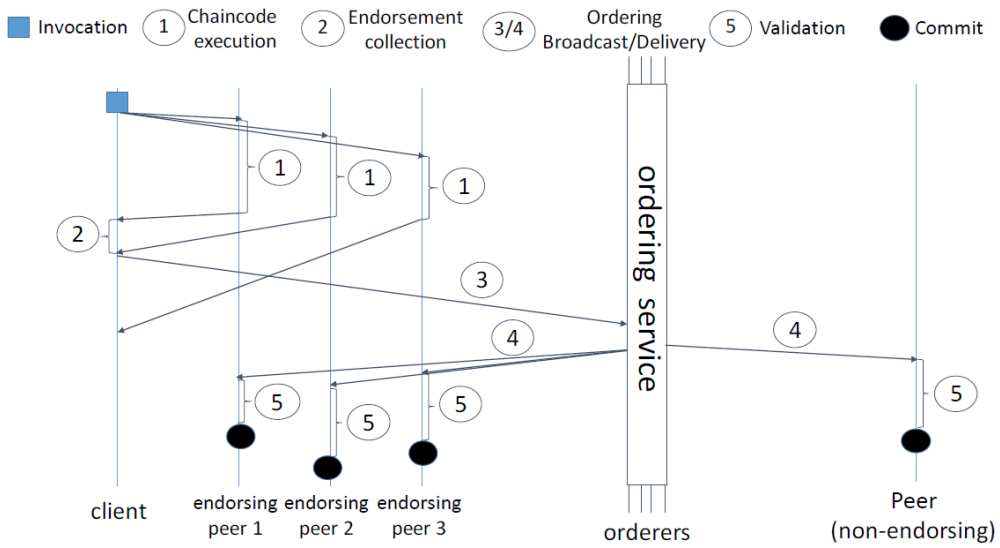

Figure 4.4: Hyperledger Fabric transaction flow

- A nonce;
- The transaction identifier, consisting of the client identifier and the nonce.

The endorsing peers simulate executing the proposed transaction according to the chaincode definition. Without synchronising with the other peers yet, the proposal is simulated against the endorser's local blockchain state. Each endorser produces a value *writeset*, which is the state update produced by the simulation, and a *readset*, which represents all the keys read during the simulation along with their version numbers. Once the simulation is complete and RW-set is built, an *endorsement* message, also containing metadata such as transaction ID, endorser ID and endorser signature is sent back to the client as a *proposal response*. The client then collects the endorsements until the required number has been reached in accordance with the endorsement policy. If and only if a sufficient number of readsets and writesets match in the messages, a proposed response will be accepted by the clients.

This process ensures the required determinism of the transactions, as the endorsement phase will not be successful if different RW-sets are presented to the client. This is an advantage over an order-execute architecture where

this situation leads to an inconsistent state between peers;

2. **Ordering phase**: when sufficient endorsements have been collected from a client, a *transaction* is created and submitted to the ordering service. This contains the transaction payload, transaction metadata, and a set of endorsements. The aim of this phase is to achieve a complete order on all submitted transactions within the channel. This is accomplished by grouping several transactions into blocks and producing an hash-linked sequence, also improving the throughput of the fault-tolerant protocol. The ordering service supports the following two operation when called by a peer:

   - *broadcast(tx)*: used to broadcast a transaction *tx*;
   - $B \leftarrow$ *deliver(s)*: called by a client to retrieve a block B using a sequence number s.

   Every ordering implementation may have its own liveness [4] and fairness [5] guarantees regarding client requests.
   A *gossip service* is additionally used to exchange data between peers. It can be scaled up and is indifferent to the execution of the consensus algorithm, enabling it to function seamlessly with both CFT (Crash Fault Tolerant) and BFT ordering services.

   It is a crucial innovation that the ordering service does not maintain any state of the blockchain or validate and executes transactions. This ensures that the consensus algorithm is highly modular and efficient.

3. **Validation phase**: during this stage blocks are distributed to peers either by the ordering service or via the gossip service. This phase is made up of three consecutive steps:

   - **Endorsement policy evaluation**: the validation system chaincode (VSCC), which is a static library included in blockchain's configuration, has the responsibility for validating the endorsement in accordance with

[4] In a distributed system is the property that guarantees progress over time. Here means that the system makes progress towards consensus, even if some nodes are unresponsive or experiencing failures

[5] In a distributed system is the property that guarantees that all participant have an equal chance of being elected as leader, participating in a consensus protocol or accessing shared resources

the endorsement policy specified in the definition of the chaincode. In case the endorsement does not meet the requirements, the transaction will be considered invalid;

- **RW conflict check**: for all transactions in the block's read-write set, a sequential check is performed. For each transaction, the keys in the *readset* undergo a check against those in the current state of the ledger, as stored locally by the peer. If the version does not match, the transaction is marked as invalid and its effect does not update the world state;
- **Ledger update**: the block is appended to the locally stored ledger and the blockchain state is updated. Any update to the state is applied by writing all key-value pairs in *writeset* to the local state.

Due to its design choices, unlike other blockchains such as Bitcoin and Ethereum, Fabric's ledger contains also invalid transaction. This choice is made because the ordering service only create the block chain and the validation is done by peers. Additionally, the ability to store rejected transactions in a permissioned network, assists with network auditing in the event of malicious activities or DoS attacks.

Chapter 5

PoC implementation

After completing the design stage, during which the key components were selected, the implementation phase was carried out to demonstrate the feasibility of the proposed approach. The Proof of Concept was constructed based on the design choices previously described, through a sequential approach that involved the following steps:

- CI/CD pipeline definition using GitLab tools;
- Hyperledger Fabric network's structure definition;
- Smart contract (chaincode) programming;
- Application development;
- HSM integration.

The implementation was performed on a RHEL 8.7 Linux distribution, with its IP address referred to as *RHEL_IP*.

5.1 DevSecOps environment

A local instance of GitLab is installed on the working environment, ensuring that no service is exposed on a public network. The version used is 15.10.3-ee (Enterprise Edition). In order to fully leverage this tool's potential and to access its GUI, the command *ssh -L 9090:localhost:80 dzanfardino@RHEL_IP* is used. This allows access to what is exposed in the remote machine's port 80 through port 9090 on the local one.

The initial phase of implementing a functional pipeline involves the configuration of an executor capable of running pipelines jobs, known as *GitLab Runner*. As a self-managed GitLab instance is used, the runners must be manually downloaded and registered. A runner is an agent that runs the code within a Docker container on the host platform. A single shared runner, accessible to all local projects, is used in the implementation. Even though several of them can be installed to improve efficiency and reliability, one of them is sufficient for this project without sacrificing any required features.

Additionally, the *Container Registry* must be enabled as well. This provides a repository for application images, allowing distribution across multiple stages of SDL. It is typically used during the push phase to store an image of the application, which allows for further analysis and deployment.

WebGoat [34], a well-known deliberately vulnerable web application developed by OWASP, was used to implement the CI/CD pipeline. The stages have been designed to integrate DevSecOps principles as outlined in section 2.1. The resulting CI/CD flow, as also described in figure 5.1, is composed of four jobs:

- **Build**: when the pipeline is triggered, the web application is built including the most recent updates;
- **SAST**: using GitLab's semgrep [1] SAST analyzer, a static analysis of the code is performed. The results are stored as job artifact;
- **Push**: the built image is pushed to the container registry, following an authentication process;
- **DAST**: using the latest build image present in the container registry, a dynamic analysis of the application is performed and results are saved as job artifact.

[1]Semgrep is an open-source static analysis engine for locally finding bugs and detecting vulnerabilities.

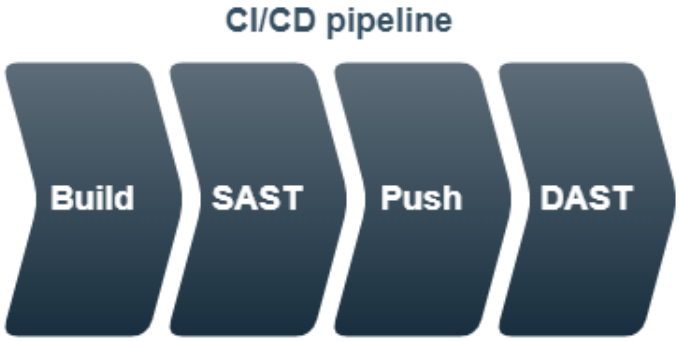

Figure 5.1: PoC CI/CD pipeline

During SAST and DAST analysis, information about the found vulnerabilities are stored in an artifact that can be accessed through authenticated APIs. More specifically, SAST analysis produces a JSON file containing an array of vulnerabilities. This array includes multiple items, each corresponding to a specific vulnerability, like the following one:

In the proposed scenario developed for the Proof of Concept, only some initial steps are present to demonstrate feasibility without adding unnecessary complexity. It should be noted that in real scenarios, as better represented in figure 5.2, multiple complex steps are typically integrated. Each step usually generates one or more artifacts in which relevant information produced in that stage is saved. Additionally, as detailed in section 2.1.1, security gates that regulate the passage from one stage to the following one are implemented. The use of security gates necessitates a premium subscription. Therefore, during development, the closest approximation to their behavior is employed. Whenever the pipeline is triggered, a minimum numbers of approvals may be required for the merging of new code to proceed. Security gates offer similar capabilities but with more detailed control for each stage of the pipeline. For example, in addition to requiring a specific number of approvals, it is possible to require also a minimum number of vulnerabilities discovered during automatic analysis.

The scope of this work was to find a way to include the information produced by the artifact and during the pipeline execution on the blockchain, providing access to necessary details for both DevSecOps team and product stakeholders. The DSO teams would likely be interested to share all technical elements obtained during development, whereas stakeholders can use relevant information to understand the development's progress or vulnerabilities and threats present in the product.

Listing 5.1: SAST result


```
{
"id": "63e8ae85fc...",
"category": "sast",
"message": "Improper Limitation of a Pathname to a Restricted
   Directory ('Path Traversal')",
"description": "A file is opened to read its content. The
   filename comes from an input parameter...",
"cve": "semgrep_id:find_sec_bugs.PATH_TRAVERSAL_IN-1:41:41",
"severity": "Critical",
"scanner": {
    "id": "semgrep",
    "name": "Semgrep"
},
"location": {
    "file": "webgoat-lessons/path-traversal/src/main/java/org/
   owasp/webgoat/path_traversal/ProfileUploadBase.java",
    "start_line": 41
},
"identifiers": [
{
    "type": "semgrep_id",
    "name": "find_sec_bugs.PATH_TRAVERSAL_IN-1",
    "value": "find_sec_bugs.PATH_TRAVERSAL_IN-1"
},
{
    "type": "cwe",
    "name": "CWE-22",
    "value": "22",
    "url": "https://cwe.mitre.org/data/definitions/22.html"
},
{
    "type": "find_sec_bugs_type",
    "name": "Find Security Bugs-PATH_TRAVERSAL_IN",
    "value": "PATH_TRAVERSAL_IN"
}
]
}
```


5.2 Network structure

Hyperledger Fabric allows the network to be highly customisable and, as explained in 4.2.1, several choices can be made. Keeping in mind the scope of

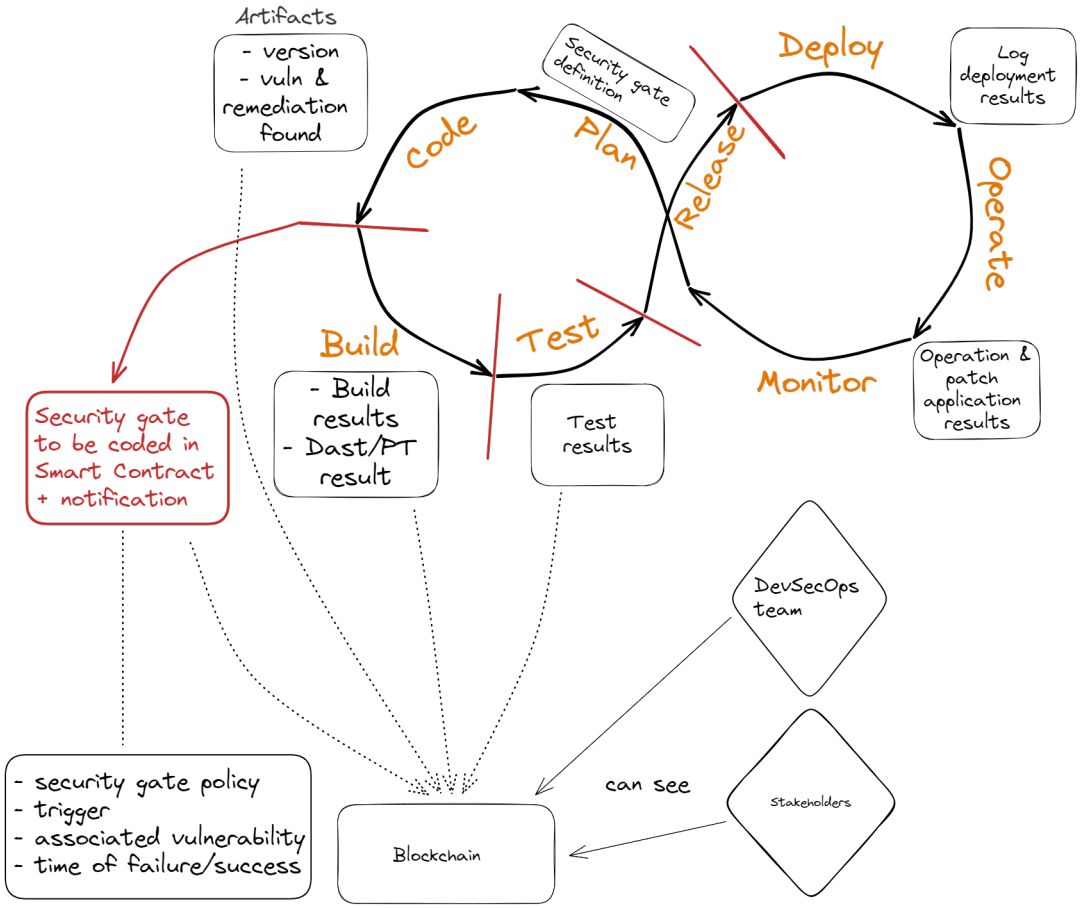

Figure 5.2: Complete pipeline artifacts and details

the project and the description made in the previous chapter, the following structure, also shown in figure 5.3, can be implemented:

- **Organisations**: this represent a logical and physical partition for the different entities interacting with the network. Two different organization are created:
  - **DSO team**: they can update pipeline implementation and, according to defined policies, perform required approval;
  - **Stakeholders**: they have visibility over current status of the project, including latest pipeline run and all found vulnerabilities.

  Each organisation has its own certificate authority, which releases and manages PKI-based certificates for network member organisations and

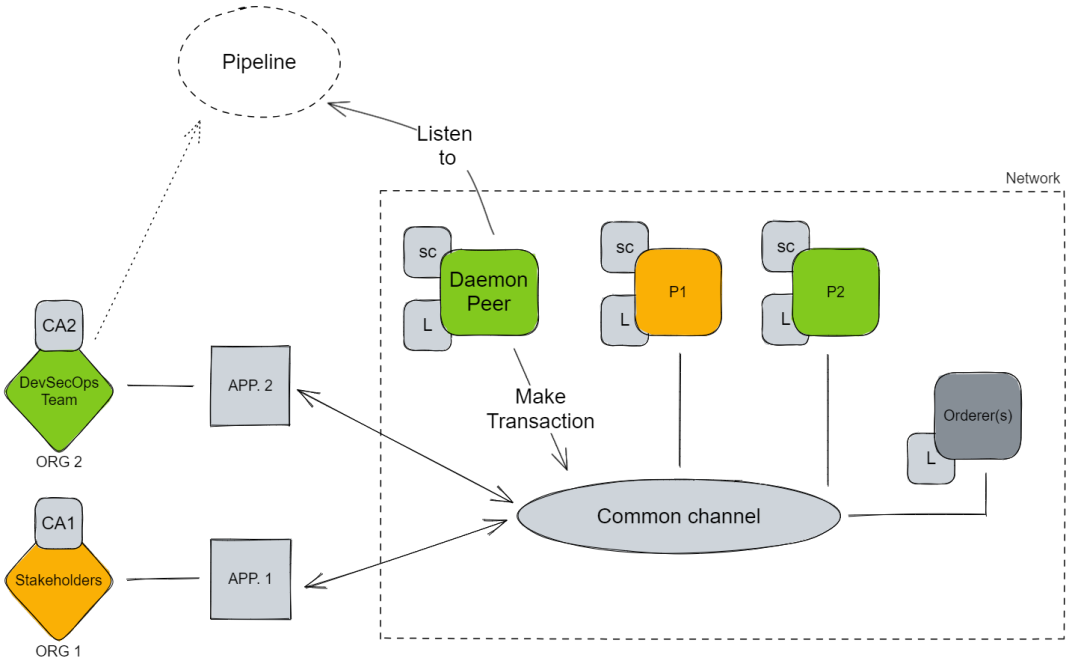

Figure 5.3: Proposed network structure

their users. Each CA is created based on specific configuration files that define its characteristics. Some of the most significant ones are the following:

  - TLS has been enabled for communication with network members. The *NoClientCert* option has been selected, which means that no client certificate should be requested during the handshake;
  - Details regarding certificate issuing and signatures are specified, such as the validity period for each released certificate, as well as permitted usage and values for certificate signing requests;
  - BCCSP (BlockChain Crypto Service Provider) section determines which cryptographic library to use and, in this preliminary implementation a software-based provider is selected, which has to be used for development purposes only. The provider generates keys using a software-based PRNG (Pseudo Random Number Generator), which are then stored in specified *keystore* files.

- **Peers**: for the purpose of this project each organizations will be provided with an administrator peer, which is mandatory for each organization, and one other peer. It is unlikely that in a real-world scenario an organisation will consist of only one peer, considering that multiple peers must be

present to fully exploit the decentralized and distributed aspect of the network. Nevertheless, for the development of this PoC it is sufficient to have one peer for each organisation to assess the validity of the proposed network composition. Similar to CAs, in this initial implementation peers also use a software *keystore* file to store and retrieve needed keys and certificates.

An additional peer belonging to the DSO Team organisation, named *"Daemon Peer"* in Figure 5.3 is also defined. This is the identity designated to invoke the chaincode when changes occur in the pipeline or when jobs result become available. An application is used to make the peer access the ledger and two main operation can be performed: ledger-query and ledger-update [35].

- **Orderer**: the proposed network implements a single orderer node. However, as also anticipated in chapter 4.2.1.1, in order to maximise infrastructure benefits, reduce bottleneck and improve security, a minimum of three nodes should be used in production environment. Nonetheless, using a single orderer node does not compromise the validity of the proposed architecture. Additional ordering nodes can be easily added by appending the respective addresses and ports in the *"orderer"* section of *configtx.yaml* file, ensuring the corresponding containers have been deployed.

  To establish agreement within the network, a suitable algorithm must be selected. Hyperledger Fabric allows for several consensus algorithm including Raft, Kafka and Solo. Ultimately a Crash Fault Tolerant (CFT) ordering service based on an implementation of Raft protocol is used [36]. This protocol adopts a "leader and follower" approach, where followers replicate the entries suggested by the leader. In case the leader crashes, a new one is elected after a timeout period. The system can sustain the loss of nodes as long as there is a majority of the ordering nodes remaining. In case three ordering nodes are present, the system can withstand the loss of one node, this leaving two active nodes.
  The *"etcd"* implementation of Raft protocol, already in use and tested in several distributed frameworks like Kubernetes and Docker Swarm, is selected as consensus algorithm for this project. Raft nodes identify each other using TLS pinning, meaning that an attacker must first discover the private key of the certificate used in the communication in order to maliciously impersonate the node. Therefore, the configuration file must include the path to the certificate, along with host and port details;

- **Policies**: the *policies* section inside *configtx.yaml* defines policies for readers, writers and endorsers at different level of granularity. Both Signatures and Implicit Meta policies are defined for the proposed network. The following one, for example, refers to Organisation 1:

```
Policies:
 Readers:
  Type: Signature
  Rule: "OR('Org1MSP.admin','Org1MSP.peer','Org1MSP.client')"
 Writers:
  Type: Signature
  Rule: "OR('Org1MSP.admin', 'Org1MSP.client')"
 Admins:
  Type: Signature
  Rule: "OR('Org1MSP.admin')"
 Endorsement:
  Type: Signature
  Rule: "OR('Org1MSP.peer')"
```

By specifying the rules above, read access is restricted to clients, peers or administrators from organisation 1, while writing privileges are granted to admin or client. Similarly,endorsement of transactions is only permitted for peers from organisation 1.

Other higher level policies, such as those outlined below, are defined at the application level. This approach ensures that whenever the underlying network structure changes slightly, such as a new organisation joining the network, those rules do not need to be completely rewritten:

```
Policies:
 Readers:
  Type: ImplicitMeta
  Rule: "ANY Readers"
 Writers:
  Type: ImplicitMeta
  Rule: "ANY Writers"
 Admins:
  Type: ImplicitMeta
  Rule: "MAJORITY Admins"
 LifecycleEndorsement:
  Type: ImplicitMeta
  Rule: "MAJORITY Endorsement"
 Endorsement:
  Type: ImplicitMeta
  Rule: "MAJORITY Endorsement"
```

Several additional basic policies have been implemented to ensure proper network operation. These policies can be further customized to target specific behaviour.

In order to create a new network, the following actions need to be performed:

1. Each organisation must generate the cryptographic material, consisting of keys and certificates, that will define that organisation in the network. The organisation's CA serves as a Root of Trust and a separate Docker container for each of them is deployed. Each CA need to:
   - Enroll the CA administrator, the entity that is in charge of mancreate-Org1aging CA activities;
   - Register peer, user and admin;
   - Generate MSPs and TLS certificates for each of the previously created entities.
2. The genesis block is generated based on the information present in the *configtx.yaml* file. This block contains information related to organisations and peers. References to MSPs created in the previous step are then used to create the channel MSP. Subsequently, a Docker Compose file is used to define and deploy the peers and ordering service inside a common channel;

When these operations have been successfully completed, the network is properly configured and running, ready for chaincode installation.

5.3 Chaincode development

The chaincode, as discussed in Section 2.2.4 is responsible for handling the mutually agreed business logic among the members of the network. In the context of Hyperledger Fabric, the code can be written in *Go*, *Node.js* or *Java*, and runs in a separate process from the peers. Each programming language provides the same functionality for accessing the chaincode. In this project, *Node.js* has been selected for development.

By considering the implemented CI/CD pipeline, and also a more complete scenario as shown in Figure 5.2, Figure 5.4 shows the structure that has been defined as the base asset stored in the blockchain.

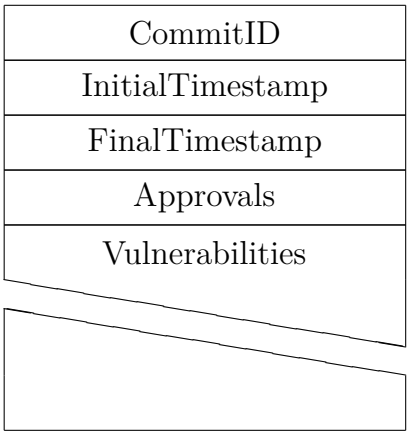

Figure 5.4: Implemented base asset structure

More specifically:

- **CommitID**: for the purpose of this PoC, it is assumed that one single project is shared between the DSO Team and stakeholders. However this assumption will not prevent the demonstration of the concept's feasibility in this project as a more general use case can be easily established by slightly modifying the proposed base asset. This field contains the CommitID of the project, a unique identifier that represents an interaction with the pipeline. All other information present in the proposed structure are linked to a specific CommitID and some may change based on action performed by DSO team members;

- **InitialTimestamp and FinalTimestamp**: considering Figure 5.2 it is noticeable that is important to save also timestamps related to security controls that are automatically performed by the pipeline. This allows to monitor when the event that triggered the pipeline occurred and how long the analysis lasted;

- **Approvals**: as also previously stated, one of the main objective of this project is to integrate blockchain and DSO through the implementation of smart contracts to increase security and ensure the necessary liability for the action performed on the pipeline. For this reason the list of all users

that, according to the specified policies, approve the incoming changes and accept the found vulnerabilities are stored. This field consist of an array of *{userID, action}* objects, where the *"action"* parameter can either indicate approval or rejection;

- **Vulnerabilities**: this section includes the information related to the vulnerabilities found during security analysis such as SAST and DAST. Due to the typically large scale of enterprise project, vulnerability lists can be quite lengthy. By looking at a possible vulnerability report artifact in Listing 5.1, we can select the relevant information to store to increase the network throughput and reduce the information replicated by each peer. Those artifact are provided by CI/CD pipeline as array of JSON objects, each containing a single identified vulnerability. For every object the following fields are saved:
  - **Message**: a short description of the vulnerability;
  - **Severity**: the severity level of the vulnerability, depending on the classification can be low, medium or high;
  - **Location**: the specific line of code in which the vulnerability is found;
  - **Name**: an identifier representing either the specific vulnerability type or the class to which it belongs, according to a well-defined standard such as CWE (Common Weakness Enumeration) which provides a standard categorisation for known software weaknesses or CVE (Common Vulnerabilities and Exposures), which define a unique identifier for the found vulnerability.

Through the use of the *fabric-contract-api*, a high-level API for smart contract development provided by Fabric, it is possible to write the required logic. The context *"ctx"* is passed, together with required parameters, to each implemented chaincode function. Using the provided SDK it is possible to:

- **Invoke** transaction: the purpose of this operation is to modify the ledger state, which involves writing new data to the blockchain. Specific policies governing this operation have been defined and explained in previous subsection. These transactions must go through the entire consensus mechanism of execution, ordering and validation, and consensus must be reached for the transaction to be effective;
- **Query** the ledger state: this operation is used to retrieve data from the blockchain ledger without making any update to the current ledger's

state. It is a read-only operation which does not involves any consensus mechanism. As a result it is faster and less resource-intensive compared to the *"Invoke"* operation. This can be executed immediately by the developed application, as a peer can immediately answer with the required information by consulting its own copy of the ledger.

The chaincode includes the following functionalities that network members can invoke:

- **InitLedger**: this function is used to initialize the ledger state. In the proposed scenario an empty state is created at the beginning since no information is needed when the network is deployed, all the required assets will be created when pipeline runs and is modified;
- **CreateAsset**: this is the function that is called to initialize the base asset using data available at the beginning of the CI/CD pipeline run, such as the *CommitID*, the InitialTimestamp and the array of required approvers;
- **ReadAsset**: this function returns the asset stored in the current world state given its *CommitID*;
- **UpdateAsset**: according to the parameters provided to this function, the asset with the corresponding *CommitID* is updated;
- **DeleteAsset**: this function deletes the asset stored in the current world state given its *CommitID*;
- **AssetExists**: this function, like the "ReadAsset" function, performs a query operation. However, its purpose is to determine whether the asset currently exists or not. Developed applications typically uses this as a service function to perform controls prior invoking a transaction;
- **GetAllAssets**: this function returns all the assets stored in the ledger;
- **GetHistoryForKey**: This function returns an array of all the modifications that an asset has undergone given a specific *CommitID*. It is especially useful for audit purposes to obtain the full history of a particular asset.

Once the definition of the chaincode has been completed, it is possible, from a blockchain network point of view to continue the steps presented in Chapter 5.2 in order to proceed with chaincode lifecycle and install the chaincode on the peers. More specifically at this stage it is needed to:

3. **Package** the chaincode: compress the developed chaincode together with other information in order to send it to interested network members;

4. **Install** the chaincode on each peer associated with participating organisations;

5. **Approve** the installed chaincode by the organisations in which the chaincode is being installed.

If all preceding steps have been successfully completed, it is possible to commit the installed chaincode on the peers. At this stage peers and applications can invoke the chaincode.

5.4 Application development

To interact with an installed chaincode, an application follows the procedure presented in Figure 5.5.

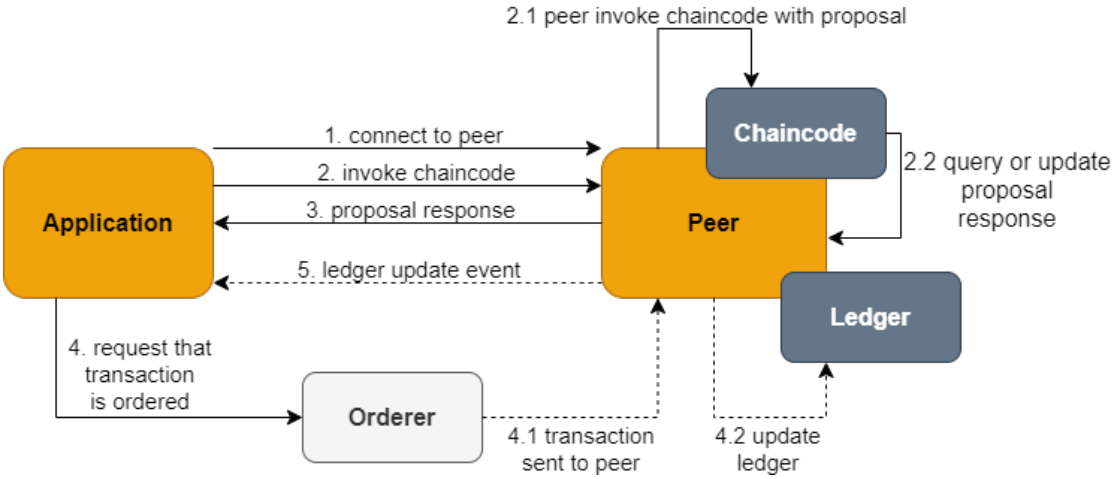

Figure 5.5: Application and peer interaction

More specifically for both Invoke and Query operations, the initial steps are the same: the application connects to the Peer and invokes the chaincode. This results in the generation of a query or update proposal response which is sent back to the application.
In the event of transaction invocation, in which changes to the ledger are made, additional steps are required. In step 4 of the diagram in Picture 5.5 the application collects all the responses and builds a transaction, which is then sent to the ordering service. The ordering service receives the transactions from the network

and in step 4.1 distributes these to all peers. The Peer must validate the received transaction before committing to the ledger state in step 4.2. An asynchronous event notifies the application that the operation has been successfully completed.

All the previous steps are hidden behind the provided SDK, that can be used to develop an application that interacts with the existing network. The server that has been developed should operate as an oracle, daemon service that unnoticeably relays information received from the pipeline into the blockchain. This task is highly complex and a major topic in current research activities. While it is directly related, it was not the primary focus of this thesis work and was therefore simplified for clarity of the proposed structure. Despite the simplifications, the main security mechanism has been implemented to ensure secure communication and provide an high level of trust in the service.

With reference to Picture 5.3 the aim of the proposed server is to enable communication between pipeline events and the blockchain network. In order to do so, it is possible to enable and configure GitLab webhooks [37], which are HTTP callbacks triggered by pipeline events that can transmit information to a configured URI endpoint. As all the development was performed locally on the same machine, the pipeline webhooks can communicate with the server through a designated port on the loopback interface.

When configuring webhooks, it is possible to specify which events to subscribe to within the project, such as push, pipeline, job, deployment or merge request events. The key events that can provide the required information for this project are essentially merge request, pipeline and job events, which send data related to changes and events happening when the pipeline is triggered. Pipeline events for example are triggered whenever there is a change in the status of the pipeline and the payload of the HTTP POST contains, among other information, the following useful ones:

- The reason that triggered the pipeline, such as a merge request;
- All the information that can be used to identify the project, such as the project ID, the branch, the name and the URL;
- Pipeline and jobs identifiers, together with timestamp related to each job and their status. For each of them, information about the artifacts are also reported.

Similarly, job events provide more detailed information on individual job statuses

and activate upon a job's state change.
Merge request event instead, among other actions, are triggered when:

- A new merge request is created by a user;
- An existing merge request is updated, approved, unapproved, merged, or closed;
- An individual user adds or removes their approval to an existing merge request.

These type of callback can provide information on the merge request itself and on the required approval status.
The information received from the webhooks can be used to populate the base asset data. Along with this, it is also possible to retrieve, using the GitLab API, the artifact produced by the pipeline jobs, such as SAST and DAST report, as the projectID and jobID are provided by the webhooks, which also notifies when these reports are available.
All the interactions between the server and the pipeline require authentication through a private access token, which must be included in the *PRIVATE-TOKEN* header of each request. SSL verification can also be enabled for outgoing HTTP requests, although it has not been enabled for this project, as self-signed certificate cannot be verified. Nonetheless, this does not invalidate the proposed schema, since authentication mechanism are in place to protect each call.

The developed server must to listen for incoming webhooks and retrieve the necessary information either from the payload of the HTTP callbacks or invoking GitLab APIs. The chaincode is called as described in Figure 5.5 and is updated with the latest updates coming from the pipeline. It is composed of three main modules:

- **DSOApp**: this main component contains the required routes to receive the callbacks, as well as logic for extracting and parsing necessary information;
- **DaemonUtils**: this module is responsible for interacting with GitLab to retrieve additional information need to be retrieved, such as artifact files. For instance, to obtain the SAST report, the API to retrieve the artifact is called when the analysis is completed. The received file is unzipped, processed, and only the necessary information is passed to the main module;

- **ChaincodeAccess**: this is the module responsible of interacting with the chaincode when needed. For both query and invoke transactions, it is needed to:
  - Build an in memory object with the current network configuration, called *connection profile*;
  - Build an instance of the CA client based on the information provided in the connection profile, that reflect the already established one in the network;
  - Setup the wallet that holds credentials of the application users;
  - Create a gateway instance by using the provided API, which allow for a connection to the fabric network. All the transaction submitted by the gateway are signed using the specified user credential stored in the wallet. For the proposed infrastructure, the additional "Daemon Peer" user of DSO organisation, presented in Picture 5.3 has been used;

  Once all necessary step has been completed, the smart contract can be invoked using the functionality developed and described in Section 5.3.

5.5 HSM integration

The proposed architecture makes extensive use of digital certificates and public key infrastructure to enable secure communication between all the involved actors. To enhance security of the overall system, an HSM can be used to store keys and perform cryptographic operations as described in Section 2.3. The default implementation for CAs, Peers and Orderers binaries uses a software-based cryptographic library and a software file-based key store. This preference is due to its fast generation of pseudo-random values for keys and its ability to be employed in a testing environment.

If stricter security measures are required, particularly in an enterprise scenario like the one proposed in this thesis, it is possible to integrate an HSM in the network. Security Reply S.r.l provided a Thales SafeNet Luna A750 [12] which possess the characteristic described in section 2.3. In order to integrate an HSM with Hyperledger fabric the following actions are required:

1. Create a partition on the HSM, either dedicated or shared by multiple applications. The first option is the one that has been used in this project and is safer;

2. After downloading the most recent binaries and Docker images, binaries have been recompiled with PKCS#11 support enabled. This operation needs to be performed with fabric-ca-client, fabric-ca-server, peer and orderer;

3. Modify Dockerfiles for each network component to:

   - Include the necessary libraries and configuration files;
   - Create a group with the same group ID as the one enabled on the host machine to access the HSM and add the user running the container to the newly created group.

4. Modify the BCCSP section of all the configuration files to include the following information:

```
BCCSP:
  Default: PKCS11
    SW:
     ...
  PKCS11:
     Library: /usr/safenet/lunaclient/lib/libCryptoki2_64.so
     Label: test_tesiHsm
     Pin: #######
     Immutable: false
     Hash: SHA2
     Security: 256
```

   This includes details on the path to the shared object that has to be used to interface with the HSM, specified in the *"Library"* section. The *"Label"* and the *"Pin"* are used to identify the partition and authenticate with the HSM. The *"Immutable"* field is used to indicate that the private key attributes can be altered after key generation. The last two fields specify hash and security level required;

5. Even if the information is redundant, the above mentioned details have been also specified in the Docker Compose used to bring up the network.

Thus, each actor connects to the HSM to perform the signature operation more efficiently and to securely generate and store the corresponding keys. All peers, orderers and CAs in the implemented PoC access the same HSM partition. In a more realistic scenario:

- Each organisation will have its personal HSM for storing keys and executing cryptographic operations for its users;
- Multiple partitions for each group of users within the same organisation's HSM can be created, improving segregation and reducing the impact in the event of a compromise.

These changes can be easily implemented by changing all required configuration files, without changing the existing network structure.

Chapter 6

Conclusion

The main objective of this master's thesis was to investigate and implement a new integration of DevSecOps and blockchain, through smart contracts. Initially, research was conducted to better understand the DSO environment, how existing frameworks work, possible actions and associated security risks. The automation of various aspects of the SDL has increased the risk in the event that these framework are improperly configured or accessed in an unauthorized manner. The most critical data that should be securely stored and for which improved accountability of actions performed is needed has been identified. Smart contracts were selected as a potential means of securing information about the agreements and policies defined between DSO team and stakeholders, thus increasing transparency and trust between the involved parties. The limited number of research on such an integration was also examined. Nevertheless the frameworks outlined in this project differs from those presented in available articles, despite some common principles.

The next task involved selecting the proper blockchain that would allow the development of smart contract in the proposed scenario. After considering several options, Hyperledger Fabric was finally selected because of its highly configurability and because it was developed specifically for enterprise scenarios. To achieve a fully functional PoC, the network structure has to be defined, composed of organisations, certificate authorities, peers and orderers, as well as policies and configurations. The chosen blockchain framework is well-suited to the suggested structure as the DSO team and stakeholders have no common interest beyond the enforcement of the initially made agreements. *Node.js* was used to develop both the smart contract, called chaincode, and an application

that enables interaction with the contract installed on the network. The developed application functions as a basic oracle, injecting off-chain data into the blockchain. The chaincode may be invoked by the network's peers or the application to update or query required information.

As Public Key Infrastructure is essential to the security of communications in the proposed structure, PKCS#11 has been leveraged to interface with an Hardware Security Module. This instrument enables efficient signature operations, as well as secure generation and storage of keys associated to X.509 certificates. As the final task to complete the PoC, binaries, Dockerfiles, Docker Compose and configuration files for peers, CAs and orderer have been updated to incorporate HSM integration.

6.1 Future improvements

As the proposed one is a decentralized infrastructure, it inherits certain limitation and criticalities. The following sections provide an analysis of the identified issues and suggested solutions. As the implementation of the proposed solutions was not the primary objective of this project, they have not been put into operation, but they can be easily implemented in the existing architecture.

6.1.1 Single Point of Failure Oracle

As already introduced in section 2.2.5, blockchain oracles are one of the main weaknesses of blockchain, undermining the adoption of this technology in complex scenarios, where off-chain data is also involved. Their trustworthiness is a key concept when data coming from external sources is injected into an environment that is considered secure by design, such as blockchain networks. The use of a single oracle node, in contrast with the proposed decentralized infrastructure, poses a security risk if the data is incorrect or manipulated.

Thanks to recent research effort, some solutions are being developed in this area, although integration is not without challenges. Adopting a proper design, it is possible to build a secure blockchain middleware exploiting Hyperledger Fabric's highly customisable network and some of its features. In typical scenarios, channels are segregated from each other, to guarantee privacy and confidentiality. Nevertheless, cross-channel communications are possible and can be leveraged to build a more secure and resilient oracle. A chaincode installed in a channel, can call another chaincode installed both in the same channel

and in a different one. If the chaincode being called is on a separate channel with respect to the one calling it, only the Query operation can be used. This operation is not involved in state validation checks, hence read-only operation are allowed.

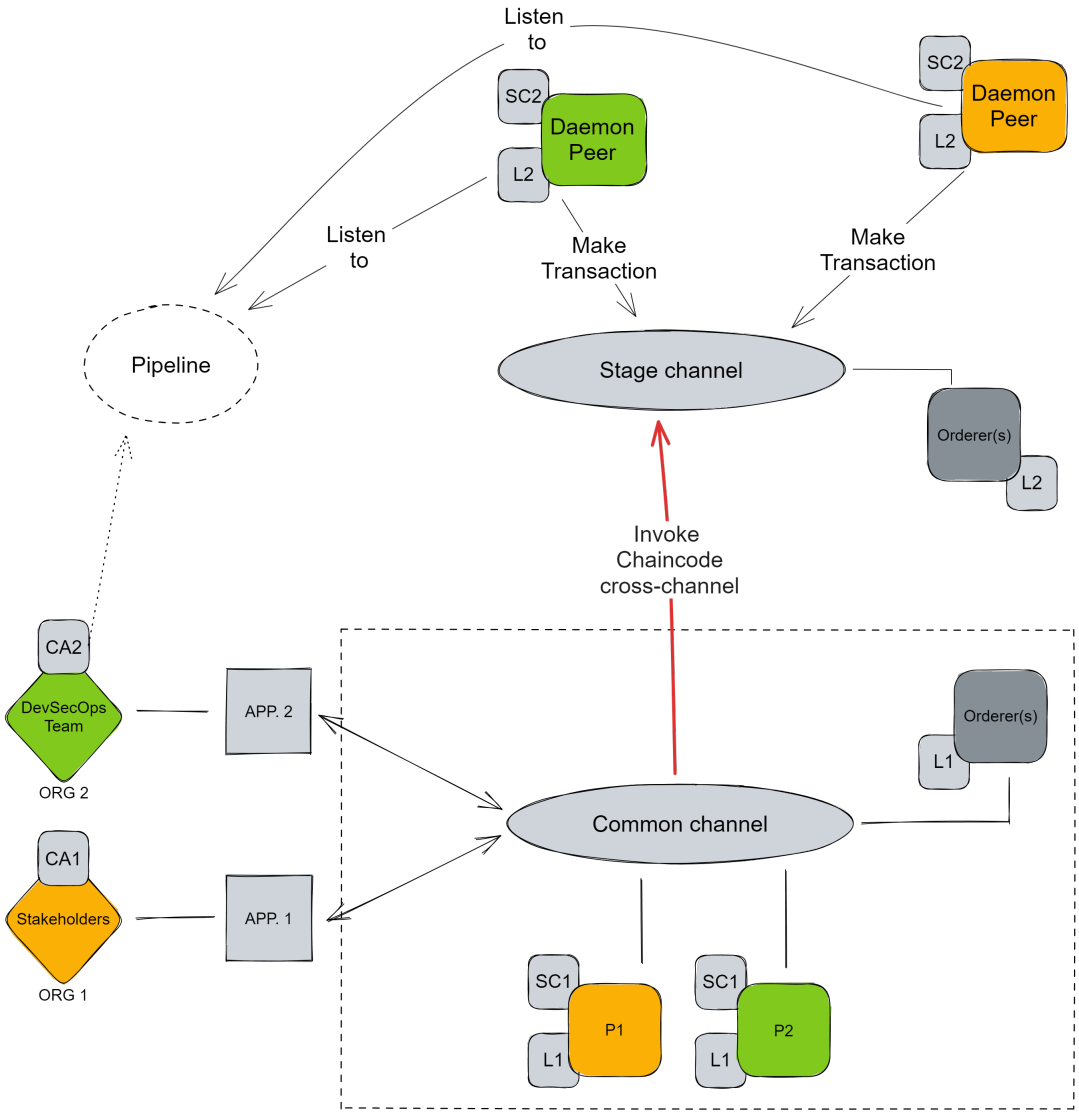

Figure 6.1: SPoF Oracle solution

The renewed structure is composed by the same organisations, but an additional channel is created. Two daemon peers, with the same functionalities as outlined in section 5.4 join the new channel, called *"staging channel"*.

Each peer belongs to a different organisation, and this is how the workflow has been modified:

1. Both daemons peer write on the staging channel relevant information coming from pipeline changes;

2. The chaincode installed on the primary channel (*"common channel"* in Picture 6.1) performs a *cross-channel* chaincode invocation by comparing data saved by each daemon peer. If the reported information is found to be consistent, only then is saved in the primary channel. In case of any discrepancies between two versions, the data is not stored on the main channel and an alarm or event is triggered.

The dual organisation structure is particularly suitable for this latest addition, as the two daemon peers added do not have any common interest beyond the provision of accurate information. This model can be extended to include multiple peer to enhance the resistance to compromised daemon peer, although the storage overhead increases with the number of peers.

6.1.2 Information reliability

Maintaining the accuracy of information derived from the source remains a challenge posed by the decentralized nature of the blockchain. Although the architecture proposed in the previous section partially addresses this issue, some additional mechanisms can be added in order to ensure the reliability of the information provided. Using the already existing architecture with some adjustments, the following steps can be implemented:

1. As git is utilized, it is possible to require signed commits using GPG (GNU Privacy Guard)[1] keys, along with either SSH keys or X.509 certificates. This ensures authenticity and integrity of the contributions;

2. Public keys belonging to the users can be stored on the ledger state;

3. The chaincode can be used to verify the correctness and validity of the signature for each received commit.

By implementing these additional measures, even if daemon peers have been compromised or are relaying modified information, it can be detected.

[1]It is an implementation of the OpenPGP standard.

6.1.3 Identity management

The developed PoC, includes the co-existence of two distinct profiles for each user:

- The identities issued by the MSP, together with the associated certificates. They are used inside the blockchain network by peers and applications interfacing with the chaincode;
- The GitLab accounts for all DSO team members and stakeholders that require interaction with the pipeline.

Although enough information to identify the entity accountable for each action on the pipeline is stored in the ledger, a deeper comparison between the two identities could be implemented in order to make this integration more comprehensive. Nevertheless, this is not a straightforward solution given that there is no guaranteed existence of one peer per user in the CI/CD environment. A mapping and linking service between the two identities can be put in place to enhance auditability of the proposed integration. Once the mapping is complete, additional actions could be executed by directly calling the chaincode instead of interacting with the pipeline. This approach would improve integration and provide more accurate auditability for the performed actions.

Bibliography

[1] United States Department of Defense. «DoD Enterprise DevSecOps Strategy Guide». In: (Sept. 2021), p. 6 (cit. on pp. 4, 5).

[2] Devarshi Singh; Varun Ramachandra Sekar; Kathryn T. Stolee; Brittany Johnson. «Evaluating How Static Analysis Tools Can Reduce Code Review Effort». In: *IEEE Symposium on Visual Languages and Human-Centric Computing* (Oct. 2017), pp. 101–105 (cit. on p. 5).

[3] Juan de Vicente Mohino; Javier Bermejo Higuera; Juan Ramón Bermejo Higuera; Juan Antonio Sicilia Montalvo. «The Application of a New Secure Software Development Life Cycle (S-SDLC) with Agile Methodologies». In: *Electronics* (2019) (cit. on p. 5).

[4] *Gitlab: vulnerability statistics*. `https://www.cvedetails.com/vendor/13074/Gitlab.html` [Accessed: (July 2023)] (cit. on p. 7).

[5] Dylan Yaga; Peter Mell; Nik Roby; Karen Scarfone. «Blockchain Technology Overview». In: (Oct. 2018). DOI: `10.6028/NIST.IR.8202` (cit. on pp. 8, 9).

[6] Yue Hao, Yi Li, Xinghua Dong, Li Fang, and Ping Chen. «Performance Analysis of Consensus Algorithm in Private Blockchain». In: *2018 IEEE Intelligent Vehicles Symposium (IV)* (2018), pp. 280–285. DOI: `10.1109/IVS.2018.8500557` (cit. on pp. 10, 11).

[7] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. «An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends». In: *2017 IEEE International Congress on Big Data (BigData Congress)* (2017), pp. 557–564 (cit. on p. 11).

[8] Miguel Castro; Barbara Liskov. «Practical Byzantine Fault Tolerance». In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Feb. 1999) (cit. on p. 11).

[9] Shuai Wang, Liwei Ouyang, Yong Yuan, Xiaochun Ni, Xuan Han, and Fei-Yue Wang. «Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends». In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 49.11 (2019), pp. 2266–2277. DOI: 10.1109/TSMC.2019.2895123 (cit. on p. 12).

[10] Ammar Hassan, Imran Makhdoom, Waseem Iqbal, Awais Ahmad, and Asad Raza. «From trust to truth: Advancements in mitigating the Blockchain Oracle problem». In: *Journal of Network and Computer Applications* 217 (2023), p. 103672 (cit. on p. 14).

[11] *PKCS#11 Terminology*. http://www.pkiglobe.org/pkcs11_terminology.html [Accessed: (August 2023)] (cit. on pp. 15, 18).

[12] *Luna Hardware Security Modules*. https://thalesdocs.com/gphsm/luna/7/docs/network/Content/Product_Overview/the_luna_hsm.htm [Accessed: (August 2023)] (cit. on pp. 15, 59).

[13] Boaz Barak; Ronen Shaltiel; Eran Tromer. «True Random Number Generators Secure in a Changing Environment». In: *Department of Computer Science and Applied Mathematics Weizmann Institute of Science* (Dec. 2003) (cit. on p. 15).

[14] *Luna Hardware Security Modules Security*. https://thalesdocs.com/gphsm/luna/7/docs/network/Content/Product_Overview/security.htm [Accessed: (August 2023)] (cit. on p. 16).

[15] OASIS Standard. «PKCS #11 Cryptographic Token Interface Base Specification Version 2.40». In: (Apr. 2015) (cit. on p. 17).

[16] *An introduction to PKCS#11*. https://thalesdocs.com/gphsm/ptk/5.9/docs/Content/PTK-C_Program/intro_PKCS11.htm [Accessed: (August 2023)] (cit. on pp. 17, 18).

[17] *User Access Control*. https://thalesdocs.com/gphsm/luna/7/docs/network/Content/Product_Overview/user_access_control.htm [Accessed: (August 2023)] (cit. on p. 19).

[18] Kimberly Connors; Abhishek Sinha; Petar Nikolic; Ivica Popovic; Ron Stokes. «Blockchain in DevOps Implementing transparent continuous delivery». In: (Sept. 2017) (cit. on pp. 21, 22).

[19] Sandip Bankar and Deven Shah. «Blockchain based framework for Software Development using DevOps». In: (2021), pp. 1–6. DOI: 10.1109/ICNTE51185.2021.9487760 (cit. on p. 24).

[20] *What is IPFS*. https://docs.ipfs.tech/concepts/what-is-ipfs/ [Accessed: (August 2023)] (cit. on p. 24).

[21] *Jenkins homepage*. https://www.jenkins.io/ [Accessed: (August 2023)] (cit. on p. 27).

[22] *GitHub homepage*. https://github.com/features/actions [Accessed: (August 2023)] (cit. on p. 27).

[23] *GitLab homepage*. https://about.gitlab.com/ [Accessed: (August 2023)] (cit. on p. 27).

[24] *Introduction - Hyperledger Fabric Documentation*. https://hyperledger-fabric.readthedocs.io/en/release-2.5/whatis.html [Accessed: (August 2023)] (cit. on p. 30).

[25] *Corda - The open permissioned distributed application platform*. https://corda.net/ [Accessed: (August 2023)] (cit. on p. 30).

[26] *What is Ethereum? The foundation for our digital future*. https://ethereum.org/en/what-is-ethereum/ [Accessed: (August 2023)] (cit. on p. 30).

[27] Tharaka Mawanane Hewa, Yining Hu, Madhusanka Liyanage, Salil S. Kanhare, and Mika Ylianttila. «Survey on Blockchain-Based Smart Contracts: Technical Aspects and Future Research». In: *IEEE Access* 9 (2021), pp. 87643–87662. DOI: 10.1109/ACCESS.2021.3068178 (cit. on p. 30).

[28] Yue Hao, Yi Li, Xinghua Dong, Li Fang, and Ping Chen. «Performance Analysis of Consensus Algorithm in Private Blockchain». In: (2018), pp. 280–285. DOI: 10.1109/IVS.2018.8500557 (cit. on p. 31).

[29] *Hyperledger Fabric Docs - Introduction*. https://hyperledger-fabric.readthedocs.io/en/latest/whatis.html [Accessed: (August 2023)] (cit. on p. 32).

[30] *Hyperledger Fabric Docs - How Fabric networks are structured*. https://hyperledger-fabric.readthedocs.io/en/latest/network/network.html [Accessed: (August 2023)] (cit. on p. 33).

[31] *Hyperledger Fabric Docs - Membership Service Provider (MSP)*. https://hyperledger-fabric.readthedocs.io/en/latest/membership/membership.html [Accessed: (August 2023)] (cit. on p. 35).

[32] *Hyperledger Fabric Docs - Policies*. https://hyperledger-fabric.readthedocs.io/en/latest/policies/policies.html [Accessed: (August 2023)] (cit. on p. 37).

[33] Elli Androulaki et al. «Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains». In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: Association for Computing Machinery, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.3190538. URL: https://doi.org/10.1145/3190508.3190538 (cit. on p. 40).

[34] *OWASP WebGoat*. https://owasp.org/www-project-webgoat/ [Accessed: (August 2023)] (cit. on p. 45).

[35] *Applications and Peers*. https://hyperledger-fabric.readthedocs.io/en/release-2.2/peers/peers.html#applications-and-peers [Accessed: (August 2023)] (cit. on p. 50).

[36] Diego Ongaro and John Ousterhout. «In Search of an Understandable Consensus Algorithm». In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. Philadelphia, PA, 2014, pp. 305–320. ISBN: 9781931971102 (cit. on p. 50).

[37] *Gitlab webhooks*. https://docs.gitlab.com/ee/user/project/integrations/webhooks.html [Accessed: (August 2023)] (cit. on p. 57).