University of Nicosia

Department of Digital Innovation

School of Business

MSc in Blockchain and Digital Currency

Security in Decentralized Finance: Smart Contract Auditing

Supervisor: Prof. Marinos Themistocleous

Dimitris Vagiakakos, Student Number: U231N0734

April 2024

Master Thesis

Security in Decentralized Finance:

Smart Contract Auditing

Dimitris Vagiakakos

Student Number: U231N0734

Abstract:

The evolution of blockchain technology has introduced a new era of decentralized services with innovations and opportunities to digitize and decentralize the world. With the rapid growth of smart contracts and Decentralized Finance Applications has also exposed many security vulnerabilities, in the smart contracts, which serve as the backbone of DeFi applications. This master thesis presents an in-depth analysis of smart contract vulnerabilities and especially in the DeFi ecosystem, based the critical need for improving smart contract auditing practices. Through a comprehensive examination of common smart contract vulnerabilities, including reentrancy attacks, integer overflows, oracle manipulations, denial of service attacks, issues in Access Control etc., this master thesis analyzes the complex security landscape of smart contract security and auditing and analyzes how best practices from traditional IT Auditing, including security policies, event logging, risk management, security awareness and incident response plans, can be implemented in the smart contract world and companies developing smart contracts in order to describe a full Audit process. Moreover, it evaluates current smart contract auditing tools and methodologies, revealing their strengths and limitations in identifying potential security and discusses the use of Large Language Models (LLMs) chatbots in the process of auditing smart contracts. Based on this analysis, this master thesis proposes a smart contract auditing framework that integrates best practices from traditional IT auditing, personalized for smart contracts and companies that are developing blockchain technologies. This security auditing framework aims to enhance the security and resilience of smart contracts, fostering trust, transparency and further reliability in companies developing smart contract solutions and Defi applications. Finally, in the context of Security Awareness Training from traditional IT Audits, this master thesis includes a series of Security Awareness videos for Greek-speaking developers in order to further improve the understanding of Smart Contract Security in Greece and Cyprus. These series of episodes cover common vulnerabilities, development tools, and vulnerability patches vulnerability patches, and other informational topics.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my professors from my bachelor studies at Department of Digital Systems of University Of Piraeus, Christos Xenakis, Demetrios Sampson and Stefanos Gkrintzalis and for their inspiration with the Blockchain Technology and of course my colleague and friend Stavros Gkinos for researching together the field of Blockchain Technology in my very first steps. Furthermore, I want to thank my colleague and friend Panagiotis Vergopoulos who as an experienced Security Consultant in conventional IT infrastructure, he discussed with me his ideas about the Smart Contract Security Framework that has been designed on this master thesis.

Furthermore, I owe my deepest gratitude to Binance and the University Of Nicosia for awarding me a full scholarship for my MSc in Blockchain and Digital Currency. This opportunity has not only been a financial blessing but also a critical milestone in my academic and professional journey, allowing me to learn more about finance, management, extending my Computer Science background, combing cybersecurity, blockchain, finance and education.

Special appreciation is owed to my Professor Marinos Themistocleous, whose invaluable guidance and support helped me to successfully complete this master thesis and contributed significantly to the analysis and methodology I followed.

Table of Contents

Chapter 1: Introduction

1.1. Background to the area under study

The coming of Blockchain technology has introduced a significant wave of decentralized digitization to our lives. Blockchain Technology with its decentralized applications called smart contracts are coming as a new upgraded and decentralized version of web, the web3. As people and companies develop decentralized applications running on Blockchains, especially for financial services, the market expands. Consequently, there is more motivation for malicious actors to investigate smart contract and try to find potential ways to exploit them to steal funds. Cybercrime, especially after the COVID-19 pandemic, is on the rise. Malicious actors are trying to find potential vulnerabilities in websites and web servers in order to steal funds or gain access in companies' information systems and networks via social engineering and phishing attacks, using malwares or ransomwares as means to demand ransom. Other more creative malicious actors even study the way that web applications work in order to find vulnerable patterns in the code that can be exploited. As expected, these kind of attacks have rapidly appeared in the Blockchain world too as the market expands. In the way to defend and enhance the security of the decentralized applications, serving as smart contracts or a set of smart contracts as entire Decentralized Finance platforms (DeFi), companies and individuals are trying to educate themselves better to be able to develop smart contracts secure by design, in the concept of designing. Furthermore, companies in order to provide more transparency to their projects, usually develop detail documentation, white papers as well as policies and procedures that they follow to prove their transparency and trustworthy environments into the Blockchain world. To achieve these, several smart contract auditing frameworks have been developed, as well as certifications. These frameworks often try to transform the security auditing controls that are applied in traditional IT Audits in normal companies into the Blockchain World. In this master thesis, in the process of implementing our own smart contract auditing framework and developing a security awareness course for developers in smart contract security and auditing, we will try to deeply describe and implement an entire IT Audit process in Blockchain companies, including controls related to smart contract development, security policies and event logging, risk management, incident response etc. We are describing deeply many attacking scenarios and common vulnerabilities in smart contracts and especially in DeFi applications. Furthermore, we explore many Defi technologies and their own potential vulnerabilities and present some attacking scenarios on them. Based on vulnerabilities and how they could be exploited, we discuss about event logging in order to identify the attacks at an early stage and try to prevent them or execute the predefined incident response plans, based on how the risks have been designed to be addressed or mitigated. As we can understand, in order to achieve these, we not only want to build strong transparency among the community and how the smart contracts are function as well as the data flows, but also how we could manage things in case something goes wrong. Security auditing frameworks, tools and security awareness training can support individuals and companies to develop better secure by design code, following best practices.

1.2. Aim

This master thesis is trying to describe a full Audit process in the Blockchain's Smart Contract World, based on the way that IT Auditing is working in businesses and critical infrastructure and try to implement these security controls in the Blockchain world and especially in decentralized applications related on Decentralized Finance. For this reason, this master thesis is following several concepts, describing:

- The governance of a company developing smart contracts,
- Developing transparency in Blockchain related Projects,
- Developing secure smart contracts following industry's best practices,
- Utilizing tools in order to identify potential vulnerabilities in smart contracts,
- Conducting Risk Assessments in the smart contracts and the access control of company,
- Event Logging in Smart Contract, Risk Management and Incident Response,
- Security Awareness of developers in related common vulnerabilities,

Therefore, as we understand, achieving security and transparency in Smart Contracts can be challenging depending on what it is being developed and how policies, procedures are designed and followed. Thus, the primary focus on this master thesis is based on these aspects:

- To analyze the current state of security within the smart contract world and especially in the Decentralized Finance ecosystem, focusing on smart contracts vulnerabilities, exploits, how these vulnerabilities can be identified and addressed before being deployed on a blockchain and how they can be patched,
- To evaluate the existing smart contract auditing practices, frameworks and tools for their effectiveness in identifying and mitigating security risks,
- To develop a comprehensive smart contract auditing and maturity framework that integrates best practices and standards from the regular IT Audit processes to the Blockchain world and DeFi applications,
- To enchase security awareness in the development of smart contract in the Hellenic and Cypriot communities by developing a series of educational videos on Smart Contract Security and useful tools that can be utilized to develop and adopt industry best security practices.

1.3. Research Questions

- What are the most common vulnerabilities in smart contracts and especially in Defi ecosystems and what patterns can be identified from past exploits in order to be used in security awareness campaigns for the developers as well as utilizing tools to identify them before deploying the smart contracts or using Artificial Intelligence (AI) to enhance the smart contract auditing process?
- How can we implement the traditional IT Audit processes in the smart contract world to provide more trustworthiness and transparency in the industry?
- How can the proposed smart contract auditing and maturity framework address the unique challenges of auditing Defi applications and how does it compare to existing frameworks and standards?

1.4. Master Thesis Outline

This master thesis is structed to systematically identify vulnerabilities in Solidity written smart contracts and Defi ecosystems. Additionally, it presents effectiveness of current auditing practices, implementing the traditional IT auditing methods in the blockchain world, while also focusing on educational outreach within Hellenic-speaking communities by designing an educational course related to Smart Contract Security and Auditing. After introducing the subject matter and outlining the research objectives in Chapter 1, this master thesis progresses to a detailed examination of blockchain technologies in Chapter 2 and the Ethereum ecosystem and its scaling solutions in Chapter 3. Then, we discuss about developing Smart Contracts in Solidity from the security-by design perspective in Chapter 4 alongside with a comprehensive review of common vulnerabilities and their impacts on smart contracts in Chapter 5 and even explore in detail Defi applications and their potential vulnerabilities and exploitation in Chapter 6. In the Chapter 7 we discuss about how we can prevent potential attacking scenarios and limit risks by utilizing Event logging in smart contracts and in chapter 8 we discuss it even further, demonstrating about Risk Assessment as well as designing Incident Response Plans. In Chapter 9 we discuss about smart contract auditing in combination with traditional IT Auditing strategies and we are utilizing tools to identify potential vulnerabilities in smart contracts as well as utilizing Artificial Intelligence (AI) to enhance the smart contract auditing process and what are the limitations of such technologies. In Chapter 10 we can find the educational course about the security awareness training for the Hellenic-speaking community with the titles of each episode of this course and the links that are available to watch. In chapter 11 we discuss for several smart contract auditing frameworks and certificates, following the industry's best practices. After evaluating these frameworks, in Chapter 12 we present the smart contract auditing framework, inspired by the way that traditional IT Audits work, with 10 categories of controls. The thesis concludes with Chapter 13, summarizing key findings, and suggesting directions for future research.

Chapter 2: An introduction to Blockchain Technologies

To start our research, we should understand what Blockchain and its features is on a basic, layer 1 level. Blockchain is a decentralized peer to peer technology (p2p) which can keep records of transaction of data, without anyone controlling it, being able to change these data. This can be achieved as blockchain uses a lot of decentralized computers called nodes, connecting each other and by using consensus mechanisms, securing the records of data from potential malicious users as well as data manipulation attacks. Each Blockchain uses the cryptographic hash value of the previous block in blockchain which usually is 256bit and it creates a chain of connected blocks. As we can understand, Blockchain is a revolutionary technology as it creates a safe environment for developing and running decentralized applications and services which cannot be modified. Blockchain keep the data of the Blockchain secure and provides a secure place without needing third parties to be trusted. Therefore, we are building a decentralized network with trust.

In a Blockchain Network, each Blockchain usually consists of:

- The data of the Block,
- The hash value of the block (usually it uses 256bit hashing algorithms such as sha256),
- The hash from the previous Block.

The first block of a blockchain it is mentioned as Genesis Block as it does not have any hash from a previous block.

2.1. Hashing

Hashing can be described as the process of scrambling raw data in a process in which the data can no longer be reproduced on its original form. Hashing tools are taking information and by using mathematical operations on the data, they create a fixed-length non reversible string which is different from the original data, called hash.

Practically, the hash value is a summary of the original data and thus the smallest change in the raw data, it could produce an extremely different hash. There is a large variety of hashing algorithms which can be used such as MD5 and SHA-2/SHA-3 algorithms.

MD5 is a well-known hashing function which is being used to check the integrity of files being downloaded from the internet. However, md5 is susceptible to some type of attack such as collision attacks. Collision attacks can be described as attacks that 2 different inputs of raw data can produce the same hash.

On the other hand, SHA-2/SHA-3 Family hash algorithms can be considered more secure in computing hashes and they offer lower risks associated with hash collisions attacks. For this reason, SHA-2/SHA-3 family algorithms are being used in Blockchains such as Bitcoin and Ethereum.

2.2. Types Of blockchain

2.2.1. Public Blockchains

Public Blockchains are the most widespread type of Blockchains since they promote transparency, and they are public networks which means that the information is available for everyone to read it and communicate in the network. Everyone has the ability to see, read and write data on a Public Blockchain. However, that does not mean anyone can modify the data of a Public Blockchain. Private keys in wallets are being used for exactly that reason. In terms of decentralization, network remains Decentralized as everyone can download a copy of the Blockchain on his own computer and run a node or even validate blocks without a third party such as a company or an organization can control it. As a result, Public Blockchains can provide security and availability as once the entry has been validated, the information which passed through the network cannot be modified or deleted. On this type of Blockchain, anyone can participate in network and verify transactions. A few examples of Public Blockchain are Bitcoin, Ethereum, Cardano, Polygon and many other. In this master thesis we would use Public Blockchains to run our Smart Contracts and simulate attacks.

1.1.1. Private Blockchains

Despite the way that Public Blockchains function, Private or Permissioned Blockchains have restrictions on who can read, create and validate transactions on a network. In Private Blockchains, not everyone is able to download a copy of the stored data from the Private Blockchain as well start producing or confirming blocks. Only central authorities (for example, a company) control the members/nodes of the network for the permissions they have over them in this particular Blockchain. This strategy ensures reliability of nodes, however the network remains centralized. Private blockchains are being used in industries like supply chain management, healthcare, finance, etc. and especially in fields where the privacy of the data is on high importance due to several regulations such as GDPR or California Security Act. Furthermore, Private Blockchains usually uses more relaxed consensus mechanism such as Practical Byzantine Fault Tolerance (PBFT), Proof of Authority or other variations, given the known identities and the lower number of nodes, In the next sections we could describe more about consensus algorithms. The most famous type of Private Blockchain is Hyperledger.

1.1.2. Consortium Blockchain

Another type of Blockchain is the Consortium Blockchain, which is a type of Blockchain between Private and Public Blockchains. Unlike private blockchains which are being controlled by one company, or a public Blockchain which information stored there is public to anyone, a consortium blockchain is run by a group of predefined selected members who make decisions together. This design ensures the security of the network and prevents members from obtaining too much power while still keeping the data private. A well-known example of this is Quorum, which is like a special version of Ethereum designed for businesses that need fast and secure ways to work together on a blockchain. These blockchains are also useful for industries like banking, supply chain, and healthcare where different companies need to share information safely and efficiently. They use special methods to agree on transactions and keep data private, making them trustworthy and quick. However, setting up and managing these blockchains can be tricky because you need to make sure everyone agrees on the rules, and you have to follow certain laws and standards. But when it is implemented correctly, consortium blockchains can be considered as a powerful tool for businesses to work together securely and efficiently.

1.2. Consensus Mechanisms

1.2.1. Proof Of Work

Proof of Work (PoW) is the algorithm that secures many Blockchain Networks, including the first version of Ethereum and Bitcoin Blockchain. For the purpose of adding a new block to the blockchain, miners need to solve a mathematical "puzzle" using their computational power. A new block is accepted by the network each time a miner successfully solves a new Proof Of Work challenge, which is so difficult that miners use expensive and specialized computers. These computers require and utilize a large amount of electrical power, that provokes the environment, which means that countries providing inexpensive electric power could mine cryptocurrencies, using PoW, in much lower cost. Consequently, these states dominate each PoW Blockchain Networks leading up in a more geographically centralized network. Proof of Work Blockchains provide adequate security only if there is a large network of miners competing for block rewards in different geographical locations. As it has been mentioned before, in case of a conflict from the chains, we have always to trust and follow the longest chain of blocks because that it has the greatest computational work of all conflicted Blockchain. Suppositionally, if the network in a large area, that hosts a considerable number of miners (also known as Mining Pools), breaks down the PoW Network is losing a significant quantity of miners that make this system safe, as a result the possibility that a hacker could gain a simple majority of the network's computational power and stage grows up, what is known as a 51% attack. Subsequently, later studies have indicated that it is possible for attacks such as selfish mining attack to be successful with an approximately of 33% of the computational power. The strategy behind a selfish mining attack involves mining new blocks hidden from the other miners, effectively building a secret chain by continuing to mine on these undisclosed blocks. Then, the self-miner reveals their own series of previously mined blocks. As the design of Bitcoin is to always follow the longest chain as the most computationally secure, in case that the selfish miner is followed by 33% of the network, it becomes possible that the network adopts the selfish miner's chain as main chain, wasting the computational power of the other legitimate miners.

1.1.1. Proof Of Stake

Proof of Stake (PoS) is a consensus mechanism by which a Blockchain Network achieves distributed consensus. In PoS-based Blockchain Networks the creator of the next block is chosen through various combinations of random selection, wealth, or age. Although the overall process remains the same as Proof Of Work, the method of reaching the "target" differs. In Proof of Stake, the role of miners from Proof of Work is replaced with validators. The Blockchain keeps track of a set of validators, and anyone holding the Blockchain's base cryptocurrency (for example, ether in Ethereum's case) can become a validator by sending a special type of transaction that locks their ether into a deposit. Validators alternate in proposing and voting on the next valid block, with the weight of each validator's vote dependent on the size of their deposit (their stake). Importantly, validator risks losing their deposit if the block they staked it on is rejected by the majority of validators.

Conversely, validators earn a small reward, proportional to their deposited stake, for every block that is accepted by the majority. The validators lock up some of their Ether as a stake in the ecosystem. Validators bet on which blocks they believe will be added next to the chain. When a block is added, the validators receive a block reward in proportional to their stake. The main advantages of the Proof of Stake algorithm are energy efficiency and security, as it encourages a greater number of users to run nodes since it is easier and affordable. The randomization process also helps to decentralize the network further geographically, as since mining pools are no longer needed to mine the blocks. Thus, PoS forces validators to act honestly and follow the consensus rules through a system of rewards and punishment. The major difference between PoS and PoW is that the punishment. For instance, in PoS is intrinsic to the Blockchain (e.g., loss of staked ether), whereas in PoW the punishment is extrinsic (e.g., loss of funds spent on electricity). Some examples of PoS Blockchains are Cardano, Polkadot and the second version of Ethereum.

1.1.2. Delegated Proof Of Stake

Delegated Proof of Stake (DPoS) is an alternative implementation of Proof Of Stake with some modifications in order to achieve better performance and less energy consumption. Delegated Proof Of Stake is a consensus algorithm which uses a reputation and a voting system to achieve consensus. Additionally, DPoS divides participants into two groups: one that elects a group of block producers and another that schedules the block production. The election process is decisive in DPoS, as it motivates the stakeholders to act in good faith to avoid losing their funds. Any participant found to be acting maliciously, risks being expelled through the voting process, due to the collective interest in securing network rewards while safeguarding their investments from the other individuals of the network. It should be noted that, in the DPoS, token holders do not directly validate block transactions. Instead, they cast votes to choose a predefined number of delegates, who then assume the responsibility of transaction validation and blockchain maintenance. This limitation on the number of validators enhances efficiency and reduces the overall energy consumption of the network. DPoS is seen as a more democratic and scalable approach to consensus than traditional methods, offering significant reductions in power and computational requirements. However, it also raises potential issues regarding centralization, as it concentrates decision-making power in the hands of a relatively small group of elected delegates that they use their personal wealth for earning more profit.

1.2. Byzantine Fault Tolerance

Byzantine Fault Tolerance (BFT) allows to distributed systems to reach consensus agreement in data value among all nodes, even in cases where might fail or provide incorrect data (unintentional errors or intentional attempts to disrupt the system). This resilience against failures in nodes is crucial for maintaining the reliability and integrity of the network. The objective of a BFT mechanism to use collective decision making in both faulty and legitimate nodes in order safeguard the network from unreliable nodes. In practice, in a BFT, it is not possible to find difference between nodes providing false or correct information however algorithms are ensuring that the system can still reach consensus. BFT mechanisms are designed to work under the assumption that a certain threshold of nodes may behave or malicious actions. However, if these malicious nodes are too many, it is very likely that the consensus would fail.

1.3. Proof of Authority

Proof Of Authority (PoA) is another consensus mechanisms in which a blockchain network reaches consensus through a set of approved validators, known as authorities, who are tasked with creating new blocks and validating transactions. Unlike PoW and PoS, where the right to write new blocks is determined by computational power or the amount of stake held, PoA relies on the identity and reputation of validators to secure the network. This approach makes PoA particularly well-suited for private or consortium blockchain networks and in enterprise environments, where trust among participants is higher. Using PoA, validators are often chosen through a vetting process where their identities are made public, providing an additional layer of accountability. The selection criteria can include several factors such as the organization's reputation, the individual's role within the organization etc. Once selected, these validators are responsible not only for creating and validating blocks but also for maintaining the network's integrity and security. However, we should consider that the centralized nature of PoA can also be seen as a drawback since it places a significant amount of trust in a limited number of validators. This can potentially lead to issues with network censorship or manipulation if the validators do not act in the best interests of the network.

Chapter 3: An Introduction to Ethereum Ecosystem

After discussing the fundamentals of Blockchain Technology, we can now move on to understanding further about Ethereum Blockchain and its capabilities serving as a decentralized computer.

Ethereum is an Open-Source public Blockchain that it works as a global decentralized computer infrastructure. Unlike Bitcoin which is the most well known cryptocurrency serving as a decentralized currency, Ethereum has the role of a decentralized computer. Ethereum is able to execute decentralized applications, called smart contracts. Smart contracts are deployed in a runtime environment, called Ethereum Virtual Machine. In the same way as other popular Blockchains ensures the validity of their data, the validity of each Ether is provided by Ethereum Blockchain, which blocks are linked and secured with Proof Of Stake consensus mechanism and hash functions, like Keccak-256. The private keys and addresses are stored in a cryptocurrency wallet. Examples of Ethereum Based wallets are Metamask and MyEtherWallet. Addresses can be used to make transactions using Ether. Due to the decentralized framework of the Ethereum, developers can build decentralized applications with built-in economic functions, as it provides high availability, auditability, transparency and neutrality, while simultaneously reduces or eliminates certain counter party risks.

3.1. Ether

As previously mentioned, Ether is the primary cryptocurrency of the Ethereum Network and serves as a reward for validators for validating blocks on the Ethereum Blockchain.

The block reward along with transaction fees, incentivizes validators to contribute to the growth of the Blockchain, making Ether essential to the network’s operation. Each Ethereum wallet secured by its own private key, maintains an Ether balance and can send Ether to other smart contracts or wallets. Ether as a measuring unit has a high value so it can be divided into smaller units for practical use. The smallest denomination of Ether is 1 Wei which is equivalent to 0,000000000000000001 Ether.

3.2. Ethereum Addresses

Ethereum addresses consist of the prefix "0x" followed by the last 20 bytes of the Keccak-256 hash of the public key derived from ECDSA (Elliptic Curve Digital Signature Algorithm) private key. The implementation of the ECDSA encryption algorithm ensures that funds can only be accessed by their rightful owners. As a byte in hexadecimal is represented by two digits, Ethereum addresses contain 40 hexadecimal digits. The format for smart contract addresses is identical, but their specific addresses are generated determined by the sender and the creation transaction nonce. It should be mentioned that user accounts and smart contracts are indistinguishable, based on their addresses, as the blockchain does not store additional data to differentiate them. Any valid Keccak-256 hash that fits the described format is recognized as valid, regardless of whether it corresponds to an account with a private key or a smart contract. In other sections, we will explore some potential attacks related to the procedures discussed in this paragraph.

3.3. Gas fees

Gas is a unit of account within the Ethereum Virtual Machine, used to calculate the transaction fees, which is the amount of ETH a transaction’s sender has to pay to the validator who includes the transaction in the Blockchain. In reality in Ethereum based networks, Gas is a separate virtual currency with its own exchange rate with Ether. The price of gas varies depending on how quickly a transaction needs to be confirmed. Higher gas prices generally lead to faster confirmation times. However, it should be mentioned that gas fees also depend on the computational effort required for the transaction. So, in practice, Gas in Ethereum is the unit that measures the amount of computational effort required to execute operations like transactions, smart contract deployments, and smart contract interactions. For these reasons, gas is a critical aspect of smart contract development and security. For instance, storing more data on the blockchain requires higher gas fees, especially for storing permanent data in the history of blockchain. Ethereum’s memory for temporary data manipulation instead of saving them on Blockchain permanently, it could cost less Gas fees. In any case, Ethereum’s protocol does not prohibit gas prices to be set to zero, however in practice it is very unlikely to be validated before they expire. In other sections, we will discuss about attacks related on gas manipulation as well as Gas optimization.

3.4. Nodes and Wallets

In Blockchain Networks, multiple nodes store the complete history of the Blockchain to observe and enforce its rules. When nodes validate a transaction, it enters a pending state until a validator chooses and then confirms each pending transaction, adding it to the main chain. Additionally, there are nodes in the network known as super nodes which work only as bridges by providing the Blockchain data to other nodes. For instance, software light wallets such as Metamask require a connection to create and send transactions. Wallets like Metamask does not store a full copy of the Blockchain locally but download only the block headers. As a result, Metamask in order to be functional, relies on full nodes such as the infrastructure provided by Infura to operate. Software light wallets do not store a copy of the Blockchain locally or validate blocks and transactions. Full nodes on the other hand, store a full copy of Blockchain. By using a full node wallet, user can be more independent and the network more decentralized. It should also be mentioned that running a full node can be more private, as the user sends the transactions directly on the network, without relying on full node providers which could be collecting information from the user such as IP addresses or even associating transactions with IP addresses. On the other hand, setting up a full node needs a lot of resources making it challenging for everyday transactions on mobile devices, especially for users engaged with multiple blockchain networks.

3.5. Transactions

Transactions are the data committed to the Ethereum Blockchain signed by an originating account, targeting a specific address (user's address or smart contract's address). Transactions on Ethereum include metadata such as the gas limit for that transaction. Creating transactions in Ethereum Network, not only we can send ether to other addresses or smart contracts, but also, we can use functions from decentralized applications called smart contracts or even deploy our own decentralized applications to Ethereum Based Blockchains.

3.6. Smart Contracts and Web3.0

Smart Contracts are decentralized computer programs which their most significant feature is their immutability. Once deployed, smart contract's code cannot be modified or change. The only way to alter the code for a smart contract is to deploy a new instance. Furthermore, it should be mentioned that the outcome after the execution of a smart contact has to be the same for everyone who executes the same smart contract. The Ethereum virtual Machine runs as a local instance of every Ethereum node, however ass all the instanced of the EVM operate on the same initial state and produce the same final state. That makes the Ethereum Network a Decentralized "world computer" running Decentralized Applications, capable of running even Decentralized Banks as decentralized applications. A set of decentralized Blockchains running their own decentralized application and interacting with one another, can create Web3.0, the decentralized version of Web that we know today with centralized companies building and distributing their products in centralized servers. It should be mentioned that smart contracts only run if they are called by a transaction. Reading from smart contracts is free of charge is it does not change the state of Blockchain, however if we want to run a function from a smart contract, we have to create a transaction which it will change the state of Blockchain thus we would have to pay for transaction fees. However, the future of Web3.0 is not limited only to Ethereum alone. Interoperability between different Blockchains and especially Ethereum compatible blockchains or Ethereum scaling solutions could play an important role in upgrading and scaling Web3.0.

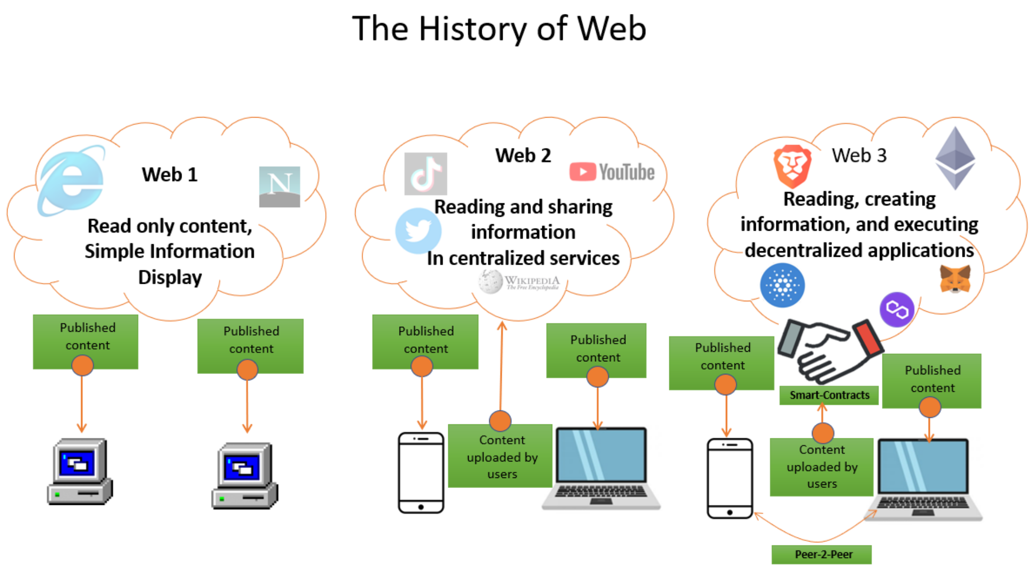

The History of Web
Web 1
Read only content, Simple Information Display
Published content
Published content
Web 2
YouTube
Reading and sharing information In centralized services
Wikipedia
Published content
Published content
Content uploaded by users
Web 3
Reading, creating information, and executing decentralized applications
Published content
Smart-Contracts
Published content
Content uploaded by users
Peer-2-Peer

Figure 1: The History Of Web as a graph.

3.7. Solidity

Solidity is an object-oriented, high-level language, explicitly for writing smart contracts with features to directly support execution on various Blockchain platforms and especially, in the decentralized environment of the Ethereum based Blockchains. Solidity was influenced by C++, Python and JavaScript program languages and is designed to target the Ethereum Virtual Machine (EVM). The main "product" of the Solidity project is the Solidity compiler (solc) which converts programs written in the Solidity language to EVM byte code. The language is still in continuous transformation and things may change in unexpected ways to resolve such issues. Solidity offers a compiler directive known as a version pragma that instructs the compiler that the program expects a specific compiler (and language) version. The Solidity compiler reads the version pragma and will produce an error if the compiler is incompatible with the current version pragma. Also, it should be mentioned that minor updates of Solidity (For example, from 0.8.0 to 0.8.24) are compatible with their main version (0.8.x.). Each Pragma directives are not compiled into EVM bytecode.

3.8. Metamask

Metamask is a lightweight wallet available for Android, iOS and desktop web browsers. The web browser version of Metamask comes as a browser extension that serves as the primary user interface to Ethereum based Blockchains. Once a user installs Metamask on his web browser he can store Ether or other ERC-20 tokens and make transactions to any Ethereum address in any Ethereum Based Network. Furthermore, the wallet can also be used by the user to interact with other Decentralized Applications or even deploy his own smart contracts to Blockchain. Metamask can be connected to multiple Ethereum Based Blockchain Networks, for example to the official main Ethereum Blockchain, in Polygon, which would be explained later, Binance's Smart-Chain (BSC) or even in test networks to help the developers or testers to test and debug smart contracts to Ethereum Test Networks, before launching them to a Mainnet Networks. Examples of Ethereum Based Testnets are Sepolia, Goerli etc. Furthermore, MetaMask can be connected to a personal testing node on a local computer using tools like Ganache.

3.9. Truffle

Truffle is an Ethereum development environment offering built in smart contract compilation, linking, deployment using the Ethereum Virtual Machine for testing purposes. By utilizing truffle, companies can develop, compile, debug smart contracts before deploying them on a virtual Blockchain Network or Testnets to test it. Truffle can also connect to Ganache in order to deploy, test and debug the smart contracts on a virtual local Ethereum Blockchain environment. For supporting the developers with debugging, Truffle provides an interactive console for direct smart contract interaction, which can be invaluable for testing and debugging as well as migration scripts for easier deploys to Ethereum based Blockchain networks (mainnets or testnets).

Truffle works with NodeJS. Truffle also has been developing an extension for Solidity support in Microsoft's Visual Studio Code, one of the most famous open source code editors.

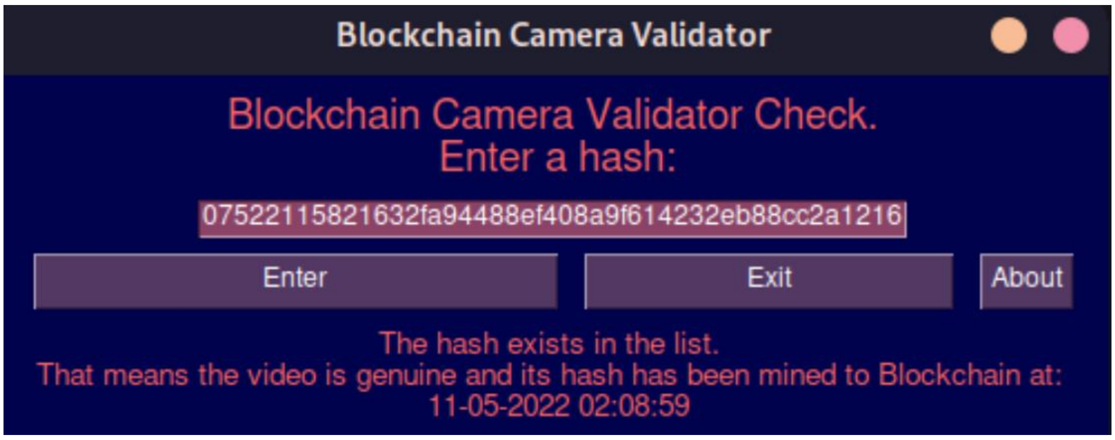
3.10. Ganache

Ethereum smart contracts are programs executed within the context of transactions on the Ethereum Blockchain. Ganache allows users to recreate a virtual Blockchain environment in their own local computers and test the smart contracts they have created locally. It simulates the features of a real Ethereum Network and it generates several virtual addresses in the local Blockchain simulator. Each address featured in Ganache owns 100 ether with their own private key, in which developers can click copy paste and get the public or private keys and test the functionality of their projects. Whenever a transaction is performed, it gets added to block in order to show the current block number in that Blockchain. Ganache can be used with Remix IDE and Metamask very easily for testing purposes and it is needed with Truffle in order to be able to deploy and test the smart contracts.

3.11. Web3 development libraries

As we have discussed before, Web3 represents a new vision and focus for web applications as we migrate from centrally owned and managed applications to applications built on decentralized protocols. Anyone can build and connect with different Decentralized Applications without permission from a central company.

There are multiple ways to interact with Ethereum Blockchain. The most famous choices are the libraries web3.js and web3.py. Web3.js which is a collection of libraries that allow users to interact with a local or remote Ethereum Node Using HTTP, IPC or WebSocket. Similarly, web3.py can be used for interaction with Ethereum Networks using Python3. Moreover, except the web3.js which is an older project working with an older, slower version of JavaScript depended to NodeJS.


Blockchain Camera Validator

Blockchain Camera Validator Check.
Enter a hash:

07522115821632fa94488ef408a9f614232eb88cc2a1216

The hash exists in the list.
That means the video is genuine and its hash has been mined to Blockchain at:
11-05-2022 02:08:59

Figure 2 A Python Application called Blockchain Camera Validator, utilizing web3.py, communicating with Ethereum Blockchain to check the validity of hashes.

3.11.1. Ether.js

Except web3.js which it is widely used for interacting with Ethereum Based Blockchains, ether3.js is another library which supports further interoperability and performance as it is only 88KB of size. Moreover, as a newer addiction to the Ethereum Development world, it supports Typescript out of the box and interoperability with other famous development tools such as Hardhat enabling further developer interaction and provider management, as the interactions with the API and much more simplified. However, being a new library comes also with challenges, as in order to remain relatively small, it needs many dependencies, leaving her less self-dependent and in situations which might not figure out yet. In any chance, in the future it may be the most widely used library for interacting with Ethereum. Finally, ether.js does not provide a web3 provider, so developers have to use custom infrastructure such as from Infura to communicate with the main or testnets of Ethereum.

3.12. Hardhat

Hardhat is another development environment for Ethereum, released in 2019 as Buidler. It offers significant improvements from other Ethereum environment tools, including full native support for TypeScript and integration with ethers.js.

Hardhat comes with Hardhat Network which is a built-in local virtual Blockchain Network in order to try and debug the smart contracts that have been developed, before deploying them on a real chain. By integrating the Hardhat Network as its local blockchain environment, Hardhat eliminates the need for external tools like Ganache for local testing, making the process much more simplified. Hardhat has also been developing an extension for Solidity support in Microsoft's Visual Studio Code.

3.13. Scaling Ethereum Network

While the Ethereum Blockchain is widely used and stands as one of the most influential Blockchains in the Blockchain ecosystem, it has several limitations in terms of the number of transactions that it can validate per minute, throughput, and the cost of transactions fees when the network is congested. These issues lead to slower transaction speeds, ending up with higher fees, restricting Ethereum's ecosystem wider adoption.

In order to achieve further mass adoption of Blockchain Networks and scalability, developers and researchers are exploring various solutions. For instance, Ethereum adoption of the Proof of Stake algorithm in its second version in the process of increasing scalability. Other proposals were including changing the size of block, change the block creation time, database partition etc. However, still we can see that the transaction fees remain very high despite these efforts.

Consequently, several solutions have been proposed in order to extend functionality while simultaneously reducing costs. These solutions fail onto two main categories:

They're proposed solutions on Ethereum's scalability is divided in two main categories:

- **On-chain Scaling**, which in practice involves an approach requiring several changes to the Ethereum protocol which can be described as a risky option as it needs numerous proposals and testes deployment in order to avoid any irregular behavior that could be destructive. These solutions are known as layer 1 scaling solutions.
- **Off-Chain Scaling** solutions, which are implemented separately from the Ethereum mainnet and do not require changes to the Ethereum protocol. These solutions are known as layer 2 scaling solutions.

Both on-chain and off-chain scaling solutions can be beneficial for enhancing the Ethereum blockchain's performance and making it more suitable for mass adoption by addressing key issues of scalability and cost.

3.13.1. Layer 2 Scaling

To address these challenges, the Blockchain community has proposed several other solutions, including sharding or even a separate Layer 2 protocols that operate independently from the main blockchain.

Layer 2 solutions are protocols which operate on top of the main Blockchain (Layer 1). Their primary role is to offload transactions from the main chain, improving the scalability and reducing the transactions fees. There are two types of Layer 2 solutions that are most common in the community:

- **State Channels,** which allow participants to conduct numerous transactions off-chain, interacting with the main chain only at the start and at the end of the transaction's sequence. For instance, Lightning Network on Bitcoin can be set as a prime example, enabling rapid micropayments,
- **Side Chains,** which are independent blockchain networks with their own consensus mechanisms and validators which in practice allow processing transactions in parallel with the main chain. While side chains can be beneficial, we should trust their consensus mechanisms as well as the bridges they use and they connect them with the main network,
- **Rollups,** which involve processing transactions off chain but posting the transaction data on-chain. There are two variates available as Rollups the Optimistic which assume that transactions are valid by default and require a challenge period during which fraudulent transactions can be disputed, and Zero Knowledge Rollups (ZK-Rollups) which as the name suggests, they are utilizing zero knowledge proofs for validation.

3.13.2. Optimism and Arbitrum Rollups

Optimism offers a an Ethereum Virtual Machine (EVM) compatible layer that supports Solidity written smart contracts, serving as a L2 Solution to scale Ethereum. Operating off-chain as a

L2 Solution, Optimism provides multiple mechanisms to prevent frauds from happening ensuring the integrity of transactions. Transactions occurring off-chain are considered valid by default. In case of a transactions being considered as invalid then the protocol issues a dispute which challenges the validity of the transaction and posting in Ethereum Blockchain in a single batch. With this approach, the transaction fees are lower, and users can make more transactions at the same time.

Arbitrum is an another L2 scaling solution from Ethereum which also executes transactions off-chain to keep costs low. Like Optimism, Arbitrum also uses rollups but it differs in its implementation details and optimizations, with those differences affecting their transaction throughput and latency.

3.13.3. Zero Knowledge Proof

Zero-knowledge proof (ZKP) is a concept from cryptography which allows one party (e.g. the prover) to prove to another party (e.g. the verifiers) that a certain statement is true, without revealing any information related on the statement itself. It can be easily described as having a crowd of people and having to proof that a person is between those people, without having to say exactly who he is. Also, it could be described as being able to prove that someone knows a secret without having to disclosure the secret by itself. ZKPs allow the verification of transactions, identities, or other information without revealing the actual data involved with. This has powerful applications in privacy-preserving technologies, especially in Blockchain as we have shown before that there is no privacy in public Blockchain at all. By utilizing zero knowledge proof technologies, it is possible to verify information without disclosing any related details. For instance, in cryptocurrencies this technology can be utilized to allow private transactions without disclosing the balance of cryptocurrencies from the sender. Furthermore, ZKPs can be beneficial for secure voting systems, where votes can prove their vote was cast without actually revealing who or what they have voted for. Finally, in terms of access control technologies and authentication, ZKPs can be sed to prove an identity or rights without compromising personal information.

3.13.4. ZKP-Rollups

A zero-knowledge rollup (zk-rollup) is a layer 2 scaling solution which enhances transaction throughput and lowers transaction costs by processing and storing the state off-chain. Furthermore, it stores transaction data in the layer 2 and only periodically posts validated batches of transactions to the layer 1 network. By managing computations and state management on these off-chain networks, zk-rollups not only can significantly reduce the load on the base blockchain, but also, they validate the correctness of transactions off-chain using zero-knowledge proofs and then submit these validated batches of transactions to the layer 1 network. As a result, data on the main chain is stored in a “compressed” format, significantly reducing transaction costs. The Polygon Network is an example of a platform that uses ZKP rollups for scaling.

3.13.5. Polygon Network

Polygon Network is another scaling solution, for Ethereum, which firstly started as MATIC Network. Polygon introduced a new Ethereum Based Blockchain using a modified version of Proof Of Stake consensus mechanism with sidechains called plasma chain which is a separate blockchain which runs in parallel with the primary blockchain and supports to transfer assets between these chains by bridges, in order to extend the limitations of Ethereum Blockchain. Developers can seamlessly develop their smart contracts for Ethereum Blockchain and then deploy them on the Polygon Network too without any further change. In practice, the Polygon as a Layer 2 network to Ethereum Network, offers great advantages by providing very low transaction fees and providing multiple sidechains to verify transactions. Polygon uses two layers to achieve interoperability with Ethereum:

- **Heimdall layer**, a consensus layer which monitors staking smart contracts being deployed on Ethereum and committing to Polygon checkpoints from Ethereum, utilizing a set of PoS Heimdall nodes to achieve that,
- **Bor layer,** which is an execution layer for the Bor nodes. The Bor Layer functions as the execution layer within Polygon's architecture, responsible for processing transactions and managing the block production process.

Moreover, Polygon in the next upgrades will be a more L2 scalable solution for Ethereum as it is in the process to support of optimistic and ZK-Rollups enhancing privacy and efficiency.

Chapter 4: Smart Contracts in Solidity

As previous mentioned, Ethereum Blockchain serves as a decentralized computer, running decentralized applications. The decentralized application running on the Ethereum Blockchain are written in Solidity programming language. Solidity is an object-oriented, high-level language which is being used for developing smart contracts, targeting the Ethereum Virtual Machine (EVM).

Smart contracts written in Solidity are applicable across various Blockchain Networks including Ethereum, Polygon, Arbitium, Binance Smart Chain. Solidity is one of the most known languages for decentralized application development and as these smart contracts can be deployed and run in many public, private and consortium Blockchains, an entire market has been created offering solutions and recommendations across various sectors, including but not limited in finance, IoT, healthcare, etc. These solutions have the potential to improve the way of living and working conditions, promoting more transparency and integrity in sharing and handling data. However, due to the rapid advancement of blockchain field, many decentralized applications are being developed without using the recommended best practices thus smart contracts and decentralized applications that are creating are low quality and are exposing their users to risks.

4.1. Basic Functionality In Solidity

Solidity offers a compiler directive known as a version pragma that instructs the compiler that specifies the required compiler versions (and language) for the program. The Solidity compiler reads the version pragma and will produce an error if the compiler is incompatible with the current version pragma. Also, it should be mentioned that minor updates of Solidity (For example, from 0.8.0 to 0.8.24) are compatible with their main version (0.8.x.). Each Pragma directives are not compiled into EVM bytecode. Version pragma is only used by the compiler to check the compatibility. In order to add version pragma, we type:

Pragma solidity VERSION_NUMBER;

If we want to ensure that a smart contract is compatible with all compiler versions above a specific one, we type:

Pragma solidity ˆ VERSION_NUMBER;

4.2. Function Visibility Specifiers:

In smart contracts, multiple function visibility specifiers are in use, depending on the desired action, such as:

- **Public:** A public function or variable is visible internally and externally within a smart contract. As we can understand, as a public function or variable, we can see the information stored in storage or state variables by calling a getter function. We can call a public function or variable using console with the command:
  await contract.FUNCTIONSNAME()

It should be mentioned that all available public functions and variables can be identified by calling **contract.abi,**

- **Private:** A private function or variable is only visible in the current smart contract. As we can understand, private functions or variables cannot be accessed externally, as we can with public functions or variables. However, in public Blockchain networks like Ethereum or Polygon, the contents of private variables can still be determined using web3 library with **getStorage()** function. An exploit based on this will be explained in later examples on this master thesis.
- **Internal:** An internal function is visible only inside of the smart contract,
- **External:** An external function is visible only outside of the smart contract.

4.3. Modifiers

- **payable:** Payable modifier is being used for functions in order to be able to receive Ether when they are being called by a user or smart contract,
- **view:** It is used in functions in order to disallow modifications of state,
- **pure:** It is used in functions in order to disallow modification or access of state,
- **constant:** it is used for state variables as its value is fixed and it can save gas costs since it does not require storage at a slot on the blockchain,
- **override:** States that this function, modifier or public state variable overrides the behavior of a function or a modifier in a base smart contract,
- **immutable**: It is used for state variables in order to allow only one assignment at construction time and is constant afterwards. It is stored in the code,
- **anonymous:** It is used for events which don't store event signature as topic.
- indexed: It is used for event parameters and it stores them as tropics,
- virtual: It is used for functions and modifiers. Virtual allows functions or modifiers behavior to be changed in derived smart contracts.

4.4. Software Package Data Exchange (SPDX)

SPDX (Software Package Data Exchange) is a standard format used to document information about the software license used in a file, especially in the context of open-source software. When applied to smart contracts, SPDX identifiers are used to clearly specify the license under which a smart contract is released. SPDX in practice, supports developers with automated license compliance and clarity using or interacting with smart contracts. We should consider that although smart contracts are typically open source, it does not necessary mean that they are free open source or they can be modified and re-used under any circumstances. For instance, a smart contract deployed as MIT License, means that the developers allow for almost unrestricted freedom to use, modify, copy or distribute. Many smart contracts are public with MIT license as the smart contracts offered by OpenZepellin which will be shown on this master thesis.

Other common licenses include GPL-3.-or-later which is a copyleft license that requires any derivative work to be open source under the same license, or Apache-2.0 which provides some limited protection against patent claims. Finally, there are many smart contracts initialized as UNLICENSED which means that the author of these smart contracts is not providing a free or open-source license for smart contract, potentially making them proprietary software.

To set up on a smart contract written in Solidity its license, we can use this identifier:

// SPDX-License-Identifier: XXX

In case of using MIT license, we can use this identifier:

// SPDX-License-Identifier: MIT

4.5. Developing Smart Contracts

For this master thesis due to the impracticality of including entire smart contracts and Defi applications, which are of extensive length, a GitHub repository has been created to host all materials related to this master thesis, including:

- Vulnerable Smart Contracts that have been created specifically for this master thesis,
- Attacking smart contracts have been developed to conduct attacks, completing challenges mentioned on this master thesis.
- The Smart Contract Auditing framework that has been designed on this master thesis.

In all common attacking vulnerability scenarios or the discusses interfaces, each smart contract is linked on GitHub for an easier navigation.

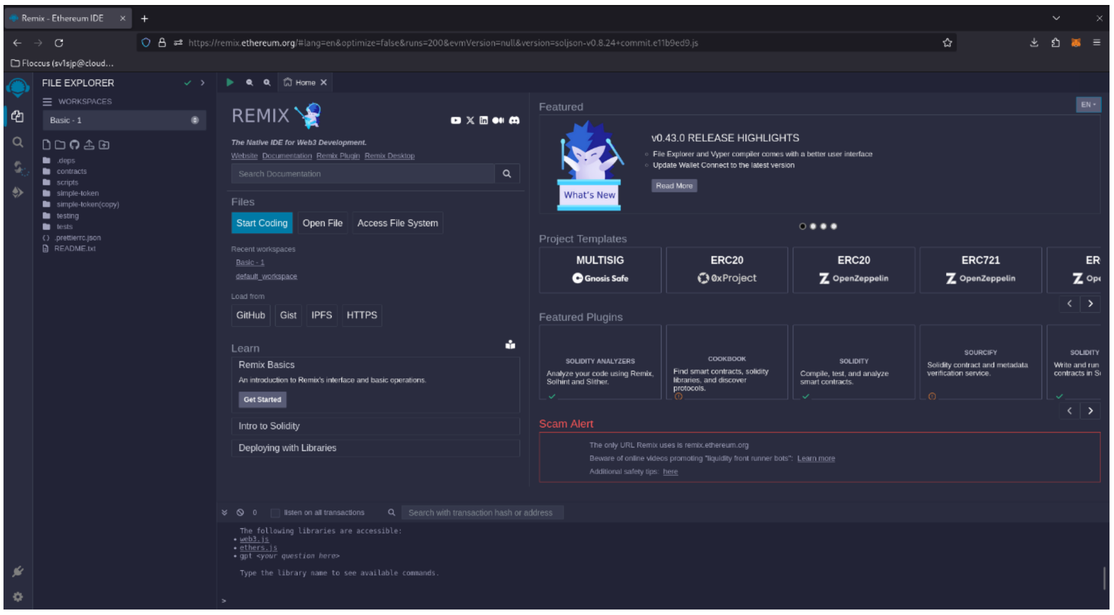
The GitHub repository can be found on this link:

https://github.com/sv1sjp/smart_contract_security_audit

4.5.1. Remix – Ethereum.org IDE

Remix is the official online IDE provided by Ethereum.org and it can be used to create, test, compile and deploy Ethereum smart contracts in the Solidity programming language. It offers several built-in Solidity compiler versions and it runs directly from the web browser. Remix IDE is extremely powerful development environment, supporting users from coding and testing, to debugging and deploying their smart contracts to Ethereum based Networks. Remix IDE can deploy smart contracts on Ethereum Mainnet, Polygon, BSC, other L2 scaling solutions or any Testnet. It should be mentioned that JavaScript VM via browser is capable of running the smart contracts from your browser without connecting to any Ethereum Based Network needed. All in all, users by using Remix IDE can even connect to their own testing node running to their computer with utilities like Ganache. In the next examples on this master thesis, we will deploy smart contracts to Sepolia Testnet as well as to local JavaScript VMs. Furthermore, Remix IDE supports extensions that further expend its functionalities and it included static analysis tools to identify potential issues in codes and recommend potential solutions. At the same time, Remix IDE offers an integrated AI chatbot which can be used to guide the developer for addressing bugs and issues in the code. Remix IDE offers a wide community and comprehensive looking to learn more about Ethereum and Solidity.

4.5.2. Writing and Deploying a smart contract using Remix IDE



v0.43.0 RELEASE HIGHLIGHTS
- File Explorer and Vyper compiler comes with a better user interface
- Update Wallet Connect to the latest version

An introduction to Remix's interface and basic operations.

Analyze your code using Remix, Solhint and Slither.

Find smart contracts, solidity libraries, and discover protocols.

Compile, test, and analyze smart contracts.

Solidity contract and metadata verification service.

The only URL Remix uses is remix.ethereum.org
Beware of online videos promoting "liquidity front runner bots": Learn more
Additional safety tips: here

```
   The following libraries are accessible:
 • web3.js
 • ethers.js
 • gpt <your question here>

   Type the library name to see available commands.

>
```

Figure 3: The main screen of Remix IDE.

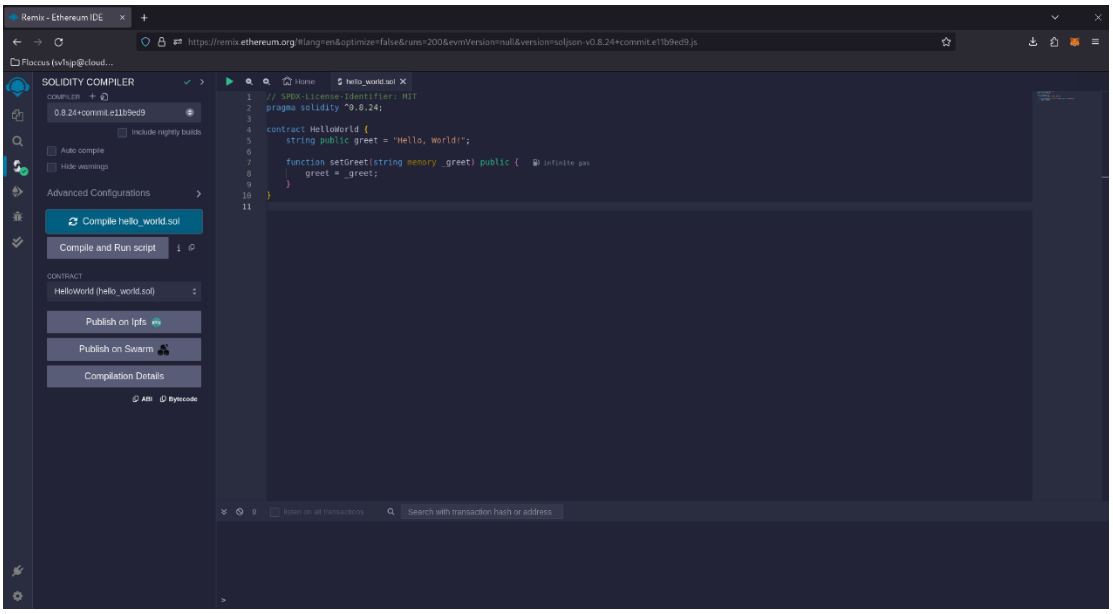
Remix IDE provides a user-friendly interface, quite similar to VS Studio Code in which developers can develop, compile and deploy smart contracts to various Blockchain Networks.

For educational purposes, we will develop a small hello world smart contract which features a function allowing users replace greeting message as they desire.

As this is the first smart contract discussed in this master thesis, we will show step by step each line of this code.

At first, we can see the SPDX License Identifier which defines the license that applies to this smart contract, in this case, GPL v3.0 or later. Then, we can see the Solidity's pragma which defines the Solidity's compiler version that it is needed in order to compile this smart contract. In this scenario, it is set in the version 0.8.24. The smart contract is named **HelloWorld** and within it, we find a public variable named **greet**, storing the phrase "Hello, World!". The public status of the variable **greet** means that its state can be directly accessed for reading. Furthermore, we notice a public function called **setGreet**() which accepts a string variable **_greet**. Again, as it is a public variable, anyone can create a transaction and call it and set the message of the greet as whatever he likes. Although the simplicity of his smart contract, it serves as an ideal example to demonstrate the basic functionality of the Remix IDE.

```
// SPDX-License-Identifier: GPL-3.0-or-later

pragma solidity ^0.8.24;

contract HelloWorld {

    string public greet = "Hello, World!";

    function setGreet(string memory _greet) public {

        greet = _greet;

    }

}
```



0.8.24+commit.e11b9ed9

HelloWorld (hello_world.sol)

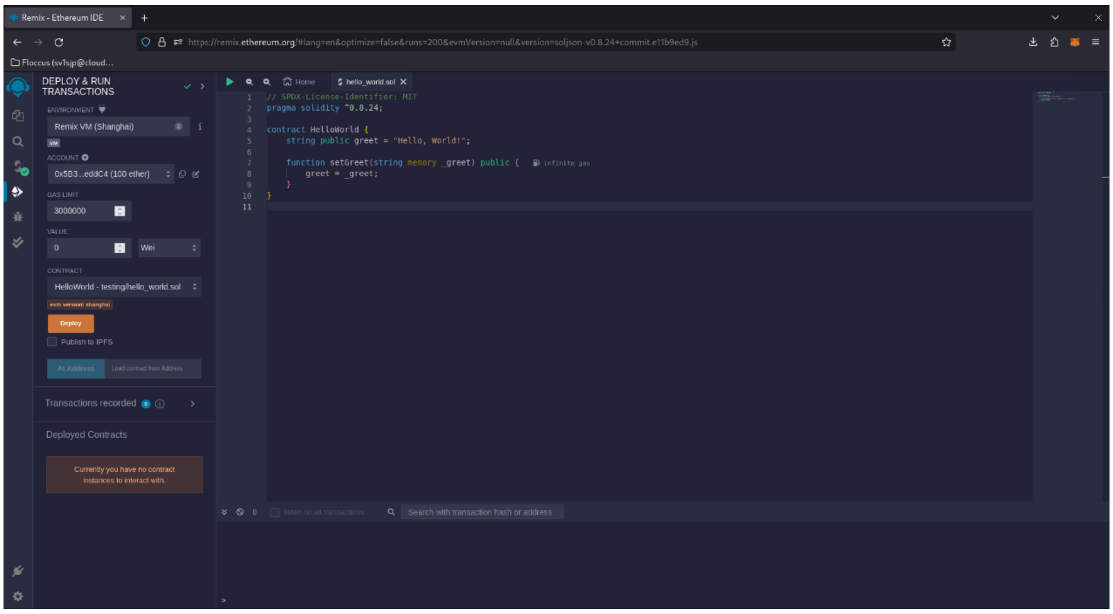
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

contract HelloWorld {
    string public greet = "Hello, World!";

    function setGreet(string memory _greet) public {    infinite gas
        greet = _greet;
    }
}
```

Figure 4: A "hello world" smart contract written in Solidity programming language.

Remix IDE offers compilers which we can compile our smart contracts and then deploy them to the Ethereum Based Blockchain Network we want. Remix IDE also shows us our syntax errors and includes an AI chatbot which can recommend us potential fixes to our issues.

Solidity version is being defined in pragma. In our scenario, the version is the 0.8.24. Remix IDE comes with a lot of versions so we will be able to try our code and syntax in any version we want.

Then, we can move to the DEPLOY & RUN TRANSACTIONS tab, which offers a lot of futures.


DEPLOY & RUN TRANSACTIONS

ENVIRONMENT: Remix VM (Shanghai)

ACCOUNT: 0x5B3...eddC4 (100 ether)

GAS LIMIT: 3000000

VALUE: 0 Wei

CONTRACT: HelloWorld - testing/hello_world.sol

evm version: shanghai

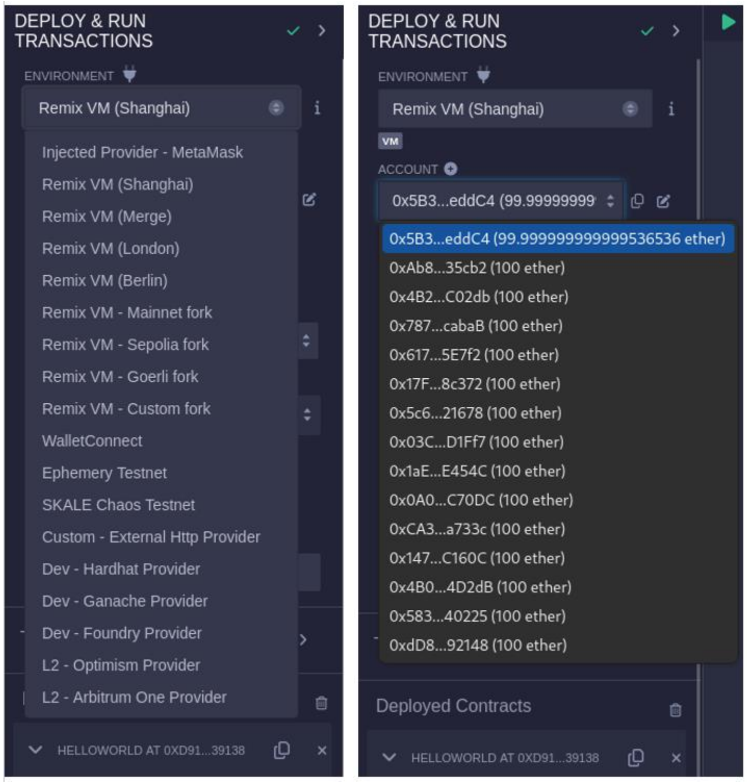
Deployed Contracts: Currently you have no contract instances to interact with.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

contract HelloWorld {
    string public greet = "Hello, World!";

    function setGreet(string memory _greet) public {
        greet = _greet;
    }
}
```

Remix - Ethereum IDE
https://remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.24+commit.e11b9ed9.js
Floccus (sv1sjp@cloud...
Deploy
Publish to IPFS
At Address
Load contract from Address
Transactions recorded
Home
hello_world.sol
infinite gas
Search with transaction hash or address

Figure 5: Deploy & Run Transactions tab on Remix IDE.

For example, in the environment we can choose the Ethereum Network we want to deploy the smart contract that we have just compiled.


DEPLOY & RUN TRANSACTIONS

ENVIRONMENT: Remix VM (Shanghai)

- Injected Provider - MetaMask
- Remix VM (Shanghai)
- Remix VM (Merge)
- Remix VM (London)
- Remix VM (Berlin)
- Remix VM - Mainnet fork
- Remix VM - Sepolia fork
- Remix VM - Goerli fork
- Remix VM - Custom fork
- WalletConnect
- Ephemery Testnet
- SKALE Chaos Testnet
- Custom - External Http Provider
- Dev - Hardhat Provider
- Dev - Ganache Provider
- Dev - Foundry Provider
- L2 - Optimism Provider
- L2 - Arbitrum One Provider

HELLOWORLD AT 0XD91...39138

DEPLOY & RUN TRANSACTIONS

ENVIRONMENT: Remix VM (Shanghai)

ACCOUNT: 0x5B3...eddC4 (99.99999999

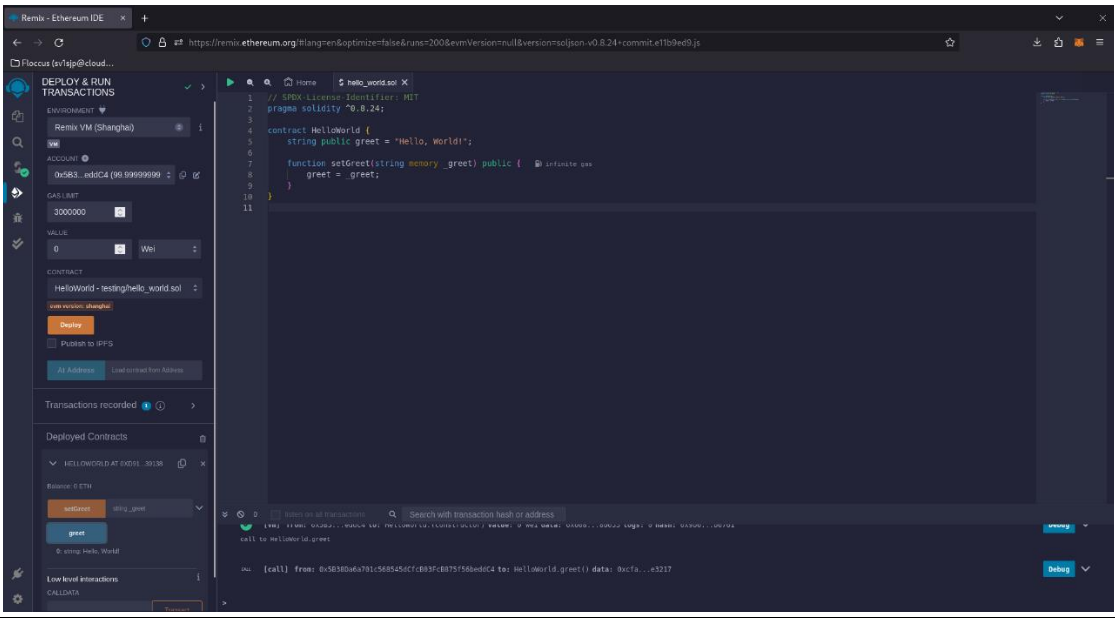
- 0x5B3...eddC4 (99.999999999999536536 ether)
- 0xAb8...35cb2 (100 ether)
- 0x4B2...C02db (100 ether)
- 0x787...cabaB (100 ether)
- 0x617...5E7f2 (100 ether)
- 0x17F...8c372 (100 ether)
- 0x5c6...21678 (100 ether)
- 0x03C...D1FF7 (100 ether)
- 0x1aE...E454C (100 ether)
- 0x0A0...C70DC (100 ether)
- 0xCA3...a733c (100 ether)
- 0x147...C160C (100 ether)
- 0x4B0...4D2dB (100 ether)
- 0x583...40225 (100 ether)
- 0xdD8...92148 (100 ether)

Deployed Contracts

HELLOWORLD AT 0XD91...39138

Figure 6: Remix IDE supports connecting with various wallets and Blockchains to test smart contracts. It also supports to deploy your smart contracts to a web browser based VM.

As we have already explained, Remix IDE supports by default a lot of Blockchain Networks and we can even connect to other Blockchains such as Polygon, Binance Smart Chain and other L2 solutions such us Optimism, Arbitrum etc. We can even set up our local simulated Blockchains with Ganache and connect to it to locally develop, deploy and run our smart contracts. But the best future is the Remix VM. Remix IDE utilizes JavaScript to create a virtual Blockchain via Browser in order to be able to deploy them for testing purposes. At the same time, it provides with many virtual JS wallets which can be used to try several use cases in testing smart contracts or testing attacks, before deploying them in the Blockchain. Remix IDE can very easily be connected to Ethereum Testnets such as Sepolia Test Network by using wallets such as Metamask.



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

contract HelloWorld {
    string public greet = "Hello, World!";

    function setGreet(string memory _greet) public {
        greet = _greet;
    }
}
```

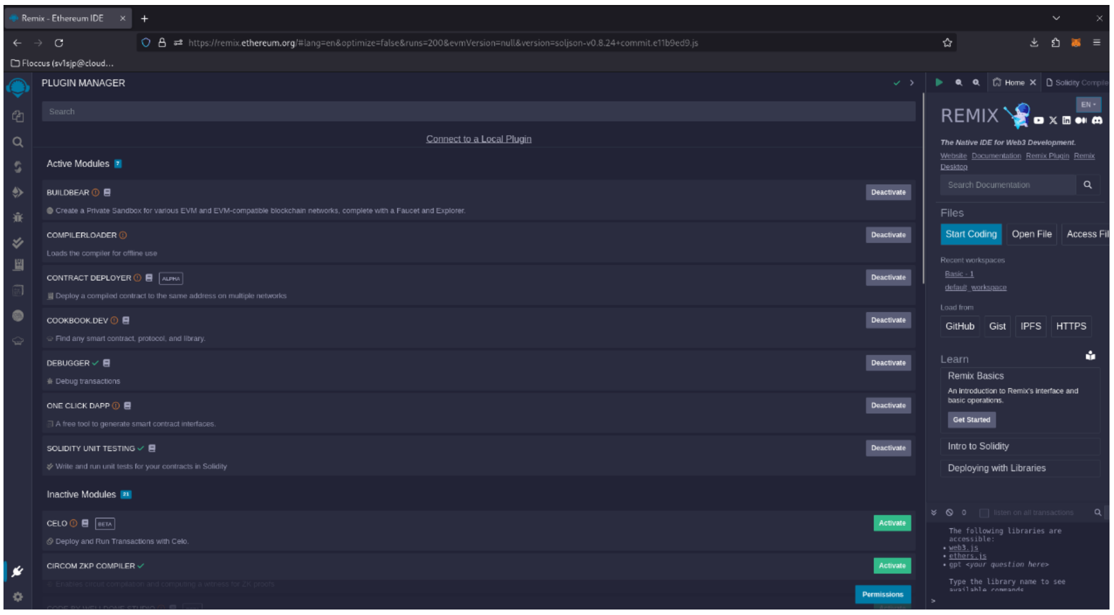
```
call to HelloWorld.greet

[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: HelloWorld.greet() data: 0xcfa...e3217
```

Figure 7: Deploying HelloWorld smart contract.

We click the Deploy button and almost immediately, the smart contract is being deployed in Remix VM. In the tab of Deployed Contracts, we can find all smart contracts that we have deployed in any Blockchain. If they don't show up, we can paste the smart contract's address on **At Address** blue button, and if we are connected on the correct Blockchain Network, the smart contract will be shown.

Remix IDE represents the functions as colorful buttons to be easier to navigate for us. The orange buttons means that we can enter data to call a function thus changing the state of the Blockchain, so in order to be executed we have to pay transaction fees. Blue Buttons represent public functions or variables which can be read from the state of Blockchain from free, almost immediately. On this example, the orange button represents the public function **setGreet()**, which is capable of replacing the Hello World string from the deployment to an another string, entered by the user.

Remix IDE also supports plugins which can extend its capabilities and support the developers to be more productive and develop more secure code. These plugins or extensions include tools related to security, debuggers, deployable audited smart contracts etc.


PLUGIN MANAGER

Connect to a Local Plugin

Active Modules 7

BUILDBEAR — Create a Private Sandbox for various EVM and EVM-compatible blockchain networks, complete with a Faucet and Explorer. — Deactivate

COMPILERLOADER — Loads the compiler for offline use — Deactivate

CONTRACT DEPLOYER (ALPHA) — Deploy a compiled contract to the same address on multiple networks — Deactivate

COOKBOOK.DEV — Find any smart contract, protocol, and library. — Deactivate

DEBUGGER — Debug transactions — Deactivate

ONE CLICK DAPP — A free tool to generate smart contract interfaces. — Deactivate

SOLIDITY UNIT TESTING — Write and run unit tests for your contracts in Solidity — Deactivate

Inactive Modules 21

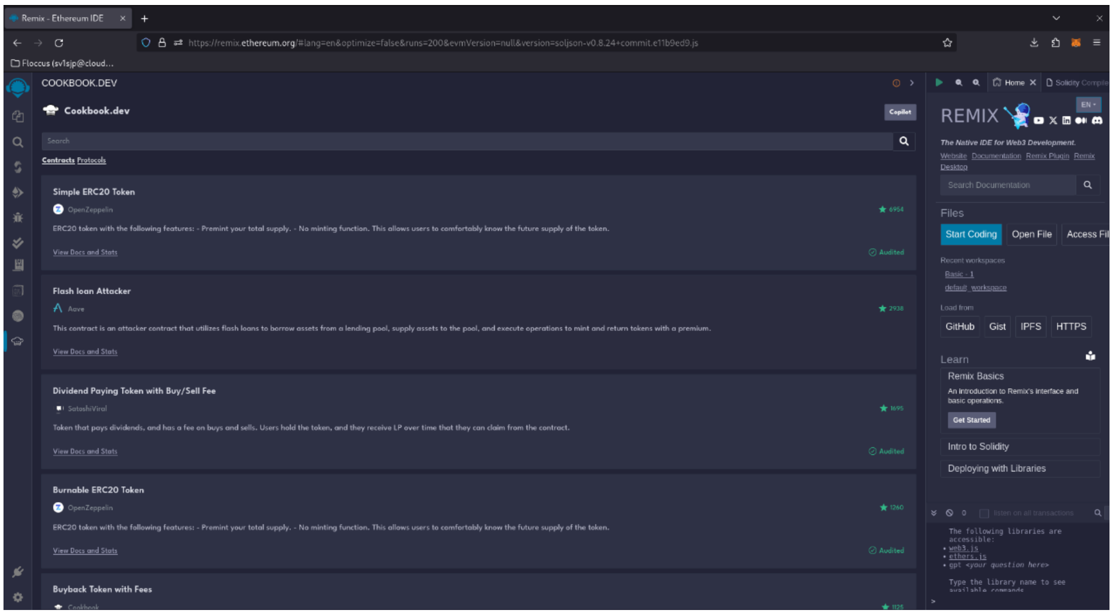
CELO (BETA) — Deploy and Run Transactions with Celo. — Activate

CIRCOM ZKP COMPILER — Enables circuit compilation and computing a witness for ZK proofs — Activate

Figure 8: Remix IDE extension modules to further scale the functionality of Remix IDE.

4.5.2.1. Cookbook.dev

Cookbook is an open source smart contract registry where developers can find Solidity based smart contracts and libraries based on Solidity Programming Language. The smart contracts that are offered from Cookbok.dev are Audited from reputable companies such as OpenZeppelin etc. Cookbok.dev plugin for Remix IDE offers a quick access tab which includes useful commonly used audited smart contracts such as a Simple ERC20 Token, NFT, Flash loan Attacker, Basic DAO and other smart contracts which many of them will be shown in the next challenges, several smart contracts from Cookbook.dev could be used.


COOKBOOK.DEV

Cookbook.dev — Copilot

Contracts Protocols

Simple ERC20 Token — OpenZeppelin — ★ 6954
ERC20 token with the following features: - Premint your total supply. - No minting function. This allows users to comfortably know the future supply of the token.
View Docs and Stats — Audited

Flash loan Attacker — Aave — ★ 2938
This contract is an attacker contract that utilizes flash loans to borrow assets from a lending pool, supply assets to the pool, and execute operations to mint and return tokens with a premium.
View Docs and Stats

Dividend Paying Token with Buy/Sell Fee — SatoshiViral — ★ 1695
Token that pays dividends, and has a fee on buys and sells. Users hold the token, and they receive LP over time that they can claim from the contract.
View Docs and Stats — Audited

Burnable ERC20 Token — OpenZeppelin — ★ 1260
ERC20 token with the following features: - Premint your total supply. - No minting function. This allows users to comfortably know the future supply of the token.
View Docs and Stats — Audited

Buyback Token with Fees — Cookbook — ★ 1125

Figure 9: A list of extentions offered by Remix IDE.

Chapter 5: Smart Contracts common vulnerabilities

After discussing the basic concepts of Solidity Programming language for decentralized applications running on Ethereum Based Blockchain Networks, we are ready to delve depper into how Solidity works, by inspecting smart contracts featuring common vulnerabilities. In this chapter we will showcase various smart contracts, developed by OpenZeppelin's Ethernaut along with others specifically developed for this master thesis. The purpose is to illustrate some well-known vulnerabilities and attacks in order to introduce useful concepts for enhancing the security of smart contracts. This section primary focuses on smart contracts for general use, but the discussed attacks can also apply to several in several Decentralized Finance applications, as we will see in the next sections too. Furthermore, the explanations of these smart contracts will help further understand how Ethereum Blockchain works and manages several concepts including memory, storage, gas fees, interfaces, implementing tokens etc.

5.1. Fallback

We have a smart contract called Fallback, which has been developed by OpenZeppelin and we are requested with claiming the ownership of the smart contract and then reducing its balance to 0. First, we need to check the smart contract's code to identify any interesting or exploitable patterns. Since this smart contract is the first vulnerable smart contract on this master thesis except the **Hello_World.sol**, it will be presented entirely. It should be mentioned again that each smart contract's name on this master thesis would be a button with a link redirecting on GitHub, OpenZeppelin or Damn Vulnerable Defi, to find the detailed smart contract's or Defi's source code.

At first, we have to discuss about the constructor. A constructor is an optional function used to initialize the state variables of a smart contract. State variables are those whose values are persistently stored in the smart contract's storage. It should be mentioned that each smart contract can have only one constructor. The constructor's code is executed once when a smart contract is deployed on a Blockchain, primarily to set the initial state of the smart contract.

owner = msg.sender;

Once the constructor has been executed, the final version of the smart contract code gets deployed to the blockchain. Bearing this in mind, during the deployment of the smart contract, as the constructor is executed first, setting the person who deployed the smart contract as the owner of the smart contract, we can check the state of the public Boolean variable **owner** using the command:

await contract.owner()

We can find the address that deployed this smart contract by typing:

contract.address

It should be mentioned that as smart contracts are being deployed on a Blockchain, they have their own unique addresses, similar to a normal user using its own wallet having its own address.

If we use the command **player** in smart contracts deployed from Ethernaut, we can find our wallet's address, as we could see it on Metamask.

As we have covered some basic topics about using smart contracts, we can now review the code and try to achieve our goals. We want to claim ownership of the smart contract, so we are trying to find something that replaces the address in the **owner** variable with our wallet's address.

We can see that there is a **receive()** external function that if we satisfy its requirements, it replaces the **owner**'s variable data with our address, thus we become the owner of the smart contract.

From Solidity's documentation, we can find out that **receive()** is a fallback function executed when transactions are made to the smart contract without specifying a function from the smart contract. The **receive()** function is triggered on calls to the smart contract with empty calldata.

This type of functions cannot have arguments or return anything. This detail can be beneficial for the next phase, so we will keep it in mind. The second requirement is to be in the **contributions** array and to have sent an amount of ether more than 0. We can check the **contributions** array to figure out if we are listed on the array calling:

await contract.contributions(player)

As expected, it returns false. So, we need to check the code again to find a way to add ourselves to the **contributions** array. We notice a payable function that requires a received amount of ether to be smaller than 0.01 ether. We can simply send 1 Wei, the smallest amount of ether that we can send to someone by typing in console:

contract.contribute({value:1})

If we check the **contributions** array again, we see our address there, returning true. So, we have satisfied the first requirement. Now it's time to send a small amount of ether again in a generic transaction to meet the second requirement and trigger the **receive()** fallback. Since the requirement is for the amount to be more than 0, we can send only 1 Wei.

Therefore, we create another transaction by typing:

contract.sendTransaction({value:1})

After the transaction is verified, we can check again who is the owner of the smart contract again and see that our address is now placed in this variable. We have successfully become the owners of the smart contract. We are ready to withdraw all the ether from it and reduce its balance to 0. As we are the owners of the smart contract, we can execute the withdraw function and withdraw all of its ether by using the command:

contract.withdraw()

And now we can check the balance of the smart contract again by using the command:

await getBalance(contract.address)

which will return a balance of 0. That means that we have successfully withdrawn all of its ether. This smart contract was an easy one that could describe the basics of how we are reviewing functions in smart contacts. In the next example we will discuss more complex issues and vulnerabilities in smart contracts.

5.2. Other issues related to constructors

In this challenge we have a smart contract from Ethernaut called Fallout.sol , which we are requested to claim the ownership of this smart contract. This smart contract is developed in Solidity's version 0.6.0 which is an older version As we have explained in the previous examples, constructor is a function which is being executed only once under the deployment of the smart contract in a Blockchain. In Solidity version 0.6.0, in order to develop a constructor function, it has to use the same name as the smart contract. By auditing In the smart contract's source code, we can notice that there is a constructor function named **Fal1out()** instead of Fallout, which is the name of the smart contract (**contract Fallout{** …**}**). As a consequence, the function **Fal1out() public payable** does not stand as a constructor, but a payable public function. As a consequence, anyone can create a transaction and call this function and become the owner of the smart contract as inside this function, there is this code:

owner = msg.sender;

allocations[owner] = msg.value;

The **owner = msg.sender;** meants that whoever creates a transaction in function **Fal1out()**, becomes the owner of this smart contract.

While in the modern versions of Solidity, the initialization that is widely used is:

constructor() public payable{}

which is more obvious and easier to avoid mistakes, it is crucial to be very curious in the development of smart contracts as a minor issue can lead to such unpredictable results. Especially in smart contracts, bugs like these can be fatal as this smart contract it is not programmed in a way that a vulnerability like this can be patched on Blockchain. That means that developers have to fix the vulnerability and then deploy again the smart contract on a Blockchain as a new smart contract, without having any direct association with the previous vulnerable version, which in case of that users have stored data on this smart contract, can create several issues related to the previous data. The new smart contract will be a new one without the data from the previous vulnerable version.

5.3. Random Number Generation in Ethereum

Coin flip is a basic game where we throw a coin and we have a 50% chance of being 0 or 1. This challenge requires 10 consecutive wins to be considered completed. In this challenge we will discuss about randomness in Smart Contracts and potential vulnerabilities based on functions calculating random numbers.

As we review the code of the smart contract, we can see that there is a function performing pseudorandom computations. Computers are unable to generate true random numbers. In order to achieve randomness, computers are trying to combine various data to produce numbers that are difficult to predict. However, if we replicate the exact steps the computer used to get the random number, we will obtain the same result. Ethereum does not provide any function for random number generation by default, leading programmers to create their own functions for randomness or use Oracles to import random numbers from outside the Ethereum Network. We will discuss further about Oracles, in the next section of this master thesis.

For the Coin-Flip game's randomness requirements, the programmer used the Safemath library (SafeMath.sol) to guard against underflows, and also it used a long uint256 number in the **FACTOR** variable.

We should consider that libraries are restricted in several ways:

- They cannot have state variables,
- They cannot inherit nor be inherited,
- They cannot receive Ether,
- They cannot be destroyed.

In the CoinFlip smart contract, there's a public function named **flip**. The function **flip** uses the blockhash method to generate a hash for a specified block. According to Solidity's official documentation, blockhash is effective only for the latest 256 blocks. Bearing this in mind, the **blockValue** variable captures the blockhash associated with the present block number. Subsequently, this value is divided by the extensive number held in the **FACTOR** variable, resulting in a Boolean result Considering our understanding of this random number generation, we explore the possibility of replicating this mechanism in another smart contract and submitting the mimicked result as our "prediction".

Therefore, we use Remix IDE to develop, compile and deploy a new smart contract on the same Blockchain as **Coinflip.sol**, the **CoinFlipAttack.sol** (as previously mentioned, OpenZepellin's Ethernaut uses the Sepolia Ethereum testnet network, so we have to use Sepolia too). In our new smart contract, we have emulated the **flip** function, but we've adjusted the blockhash technique to target the preceding block, given it's already been processed. Once compiled and deployed, our smart contract is ready to spam our cheat function 10 times, until we reach 10 consecutive wins. However, as we now understand now, there is another vulnerability that we can discuss. As validators competing to commit their blocks to Ethereum's Blockchain, miners or validators are able to exercise their judgement when building up blocks. A validator with the appropriate time, could attempt many different guesses and not commit blocks where he was mistaken. As a consequence, achieving true randomness on chain can prove be proven challenging.

Off-chain solutions based on Oracles such as Chainlink VRF (Verifiable Random Functions) represent the most considerable solutions for fair and verifiable random number generations, without compromising security. In the next section of this master thesis, we will delve into more detail for oracles and their associated vulnerabilities.

5.4. Tx.origin versus Msg.sender

In Ethereum, every transaction has an origin and a sender. The use of **msg.sender** refers to the address of the immediate sender of a transaction. It could be a user's address or another address.

On the other hand, **tx.origin** refers to the address of the externally owned account (EOA) that initiated the transaction chain.

Even if a transaction goes through multiple smart contracts, **tx.origin** will still point to the original EOA that started the transaction chain.

The problem arises when smart contracts use **tx.origin** for authorization, as it might mistakenly grant permissions to an unauthorized entity.

A potential attack scenario could involve an attacker creating a malicious smart contract with a function that, when executed, interacts with the attacker's smart contract.

Since the target smart contract checks tx.origin for authentication (and tx.origin will be the victim's address), it will assume that the call is a direct interaction from the victim. This allows the malicious smart contract to execute potentially restricted function on the target smart contact on behalf of the victim.

In this challenge, we have a smart contract called Telephone from Ethernaut.

In this smart contract, the owner can only be changed if the transaction is not initiated by an externally owned account (EOA). This is verified by ensuring that **tx.origin** and **msg.sender** are not the same.

However as we have discusses earlier, confusing **tx.origin** with **msg.sender** can lead to phishing-style attacks.

For instance, considering a scenario where an attacker creates an exploit using a smart contract named **TelephoneAttack.sol**. Within this smart contract, there's a function that directly calls the **changeOwner()** function of the Telephone contract. When this function gets executed, the **msg.sender** is the **TelephoneAttack** smart contract as it is the entity made the call. However, **tx.origin** remains the original caller of the exploit function, which is the attacker's EOA. So, the **msg.sender** and **tx.origin** are different and we satisfy the requirement.

Due to the difference between **msg.sender** and **tx.origin**, the **changeOwner()** function can be tricked into changing the owner, assuming that the call is a direct interaction, allowing the attacker to seize control. To summarize, the use of **tx.origin** for authentication or authorization should be avoided. Instead, reliance on **msg.sender** and ether establishment of clear, contract-based permission systems are recommended. If there's a need to authenticate or authorize the original EOA in a chain of calls, alternative mechanisms, such as digitally signed messages should be considered.

Maintaining a high level of awareness regarding how other smart contracts might interact with your creation and considering both direct smart contract interactions and potential interactions routed through intermediary contracts.

As we understand, using **tx.origin** in Ethereum can open the door to phishing attacks as **tx.origin** shows who started a transaction, even if there were intermediate steps. Some attackers trick people with fake smart contracts. If a person interacts with this fake smart contract, it might then communicate to the legitimate smart contract that verifies **tx.origin**. The legitimate smart contract could be misled, thinking the individual performed an action they did not intend to.

Even though **tx.origin** identifies where a transaction began, it can be used in deceptive ways. So, when we are developing smart contracts, it's a good practice to be cautious and not rely on **tx.origin** alone.

By avoiding reliance on **tx.origin** for critical operations, developers can prevent these types of phishing attacks and ensure better security for their smart contracts.

5.5. Delegation

A special message call variant, named **delegatecall()**, shares similarities with a typical message call, with the notable exception that the code at target address is executed in the context of the calling smart contract, **msg.sender** and **msg.value** do not change their values.

Consequently, **msg.sender** and **msg.value** persist unaltered. In reality, that means that a smart contract is able to dynamically load code from another address during its runtime.

However, it should be mentioned that, only the code is fetched from the called smart contract, while the storage, current address, and balance refer to the initial calling smart contract. After explaining these, let's check 2 smart-contracts from Ethernaut, utilizing **delegatecall()**.

The **delegatecall()** functions comes with the ability to invoke a function within another smart contract, allowing in practice the called smart contract to influence the state of the calling smart contract. As illustrated, the called smart contract **(Delegate)** possesses the power to modify the state of the calling one **(Delegation).**

To procced with the challenge, we first copy both smart contracts into Remix IDE. First, we deploy **Delegate.sol**, taking the address of **Delegation.sol** (which has been previously deployed by Ethernaut) as an input. To find the relevant address, we need the instance address from the challenge. After locating and copying this address, we input it during the deployment of **Delegate.sol** during its deployment. To trigger the external fallback function, which will subsequently use **delegatecall()**, we inspect **contract.abi()** to identify the signature of the **pwn()** function. Once identified, we copy the signature (in this case, "dd365b8b") and paste it into the "Low level interactions data" section in Remix IDE, to activate the fallback function. Since there's no pwn() function in **Delegate.sol**, the smart contract resorts to trigger **delegatecall()**. Executing **pwn()** modifies the state of the smart contract, making us the owner. With this, we've successfully completed the challenge.

Alternatively, we could trigger the function via the console by creating a variable:

var function = web3.utils.sha3("pwn()");

Then, initiate a fallback call using:

contract.sendTransaction({data: function});

By following this approach, we're essentially bypassing the Remix IDE interface and directly sending the transaction through the web3 console, leveraging the power and flexibility of the Ethereum JavaScript API.

Subsequent to these actions, we attain ownership of the smart contract. As we understand, **delegatecall()** carries inherent risks, being the reason of exploits in numerous historical breaches. By using the **delegatecall()**, our smart contract essentially extends an open invitation to another smart contract or library to manipulate its state at will. And if these external smart contracts can be modified to change some functionality, that means that indirectly our smart contract can also be modified.

5.6. Force Sending Ether and Destroying Smart Contracts

We have an empty smart contract called IdontwantMoney.sol without any functions, constructors or even a fallback function. Our objective is to send some Ether to this smart contract.

Ethereum smart contracts typically have 3 ways to receive ether from another address.

1) As already mentioned, by using a payable function, which does not exist in this smart contract, so we cannot use that way,
2) By receiving mining/validating rewards. A smart contract can be designed to receive block rewards. While this could be a solution for this smart contract, there is an easier way,

3) By destroying another smart contract which has ether on it by using the **selfdestruct()** method. The **selfdestruct()** method sends all ether stored on this smart contract to another address (even to addresses that are not in use) and then ending the lifecycle of the smart contract.

The third solution can be applied in this scenario. All we have to do is to move to Remix IDE and develop a very basic smart contract called **Kaboom.sol** with a payable constructor (to add ether at deployment) and a public payable attack function which will require as an input the address of the smart contract we want to send the ether to, and use **selfdestruct().**

We compile and deploy the smart contract to the Ethereum Blockchain with some ether in it. Now it's time to call the attack function previously explained and send the smart contract's address as an input. The address can be retrieved as it was shown before by using the command:

contract.address

We copy the smart contract's address and use it as a parameter in the attack function on self-destructing smart contract we just created. We are waiting for the transaction to complete, Now we can check the balance from the original smart contract address by using the command:

await getBalance(contract.address)

And now we can see that it possesses the amount of ether held by the self-destructing smart contract. Noticeably, when we use the **selfdestruct()** function and we send the amount of ether in another address, if this address is not existing or we sent it to an smart contract which hasn't any **transfer()**, **withdraw()** or even **selfdestruct()** function, it can be considered as lost forever. It is of high importance to not burn ether like this, especially in the Mainnets as we lose real world money and indirectly deflating the network's main cryptocurrency. Additionally, we should avoid designing smart contracts to allow transfers only from the owner. Someone can still use **selfdestruct()** in order to send an amount of ether to the smart contract and trigger the fallback function without being the owner.

5.7. Storage And Memory

The Ethereum Virtual Machine (EVM) uses three types of storage in order to store data on the Ethereum Network. Each account has its own storage which remains persistent across function calls and transactions. Storage in practice functions as a key-value map. Each storage slot is addressed by a 256-bit value, which is divided into 32-byte slots.

While reading public data on the Blockchain is free of cost, initializing and modifying storage on the Ethereum Network can be quite expensive, especially when we want to deploy or modify. Consequently, due to the high costs, we should minimize our persistent storge usage to only what the smart contract really needs to operate. For instance, if we want to store data on blockchain in order to prove the integrity of the data, we could store the actual data outside than blockchain and only maintain its hash value on the blockchain if we want to validate the integrity of a file.

The second data area is called memory and provide each smart contract a freshly cleared instance for each message call. Memory is linear and can be addressed at the byte level, but reads operations are limited to a width of 256 bits, while writes can be either 8 bits or 256 bits wide. It should be mentioned that the cost of using memory increases exponentially with size, making larger allocations more expensive. The Ethereum Virtual Machine is not a register machine but a stack machine, so all computations are performed on a data area called the stack. Stack has a maximum size of 1024 elements and contains words of 256 bits. As storing data in memory does not alter the history of blockchain, they don't need a lot of gas fees to be used, so we could also perform computations in memory and store data to the Blockchain only at once (in each transaction).

5.7.1. Vault

The Vault is another challenge offered by Ethernaut, which it would teach us about privacy in public Blockchain's storage. On this challenge, we have to unlock the vault to win the challenge.

First of all, we can see that we have 2 state variables: **locked** and **password**. However, we can see that one of them (**locked**) is a public variable and the other one (**password**) is private.

As locked variable is a public variable, we can easily figure out what it is stored on it by using the command:

await contract.locked()

Regarding the **password** variable, as we explained in the chapter of function visibility specifiers, as it is private, we cannot externally see directly what it is stored on this variable. Therefore, there are 2 strategies in order to "evade" and see the information stored in the password **variable**:

1) By brute-forcing the password, by trying multiple different passwords, until the correct one is discovered then unlock the vault. While it is a possible scenario, it could cost us a large amount of ether to spam the network with authentication attempts and we might fail in the end,
2) Somehow, to dump the password from the storage of the smart contract.

Ethereum is a public Blockchain, which enables everyone to read the information that is stored in the Blockchain. That means that everyone can see what exists on this variable, without having a getter public function. We can simply read the Blockchain and find out the password with a query.

Web3.js can help us achieving this, by typing in the console:

var pwd // to create a variable

web3.eth.getStorageAt(contract.address, 1, function(err, result){pwd = result})

This command in practice goes to the smart contract and reads the second slot of its storage, which is the **password**'s variable. And it will return the data in the hexadecimal mode:

0x412076657279207374726f6e67207365637265742070617373776f7264203a29

We can pretty easily copy paste this hexadecimal data to an online converter. Additionally, the hexadecimal data can be easily converted to ascii, though Web3.js by using the command:

web3.utils.toAscii(pwd)

Which will return the password: **A very strong secret password :)**

Having obtained the password in hexadecimal form, we can simply create a transaction in the function **unlock** and enter the hexadecimal value as an input by typing:

contract.unlock("**0x412076657279207374726f6e67207365637265742070617373776f72642 03a29")**

If we check the Boolean variable **locked** by typing **contract.locked()** we can see that the vault has been unlocked. From this example, we discovered the critical importance of not storing data unencrypted on blockchain networks, especially public ones, as all stored data on a smart contract is accessible to everyone.

5.8. Privacy

Now we will discuss a more sophisticated smart contract challenge related on storage. In this challenge we have to unlock a vault again from Ethernaut called Privacy.sol, but this is a more advanced one.

An audit in the code of the **Privacy.sol** smart contract, reveals several state variables and a byte32array. The constructor parses the critical data to the private variables. The unlock function serves as the interface of interacting with the smart contract and as we can understand, we have to find a way to pass the right sequence of byte16 data to unlock the vault. As it has been explained to the previous example with **Vault.sol,** the Ethereum Blockchain is public so we can inspect on its storage by typing in console:

let storage = []

let callbackFNConstructor = (index) => (error, contractData) => { storage[index]= contractData}

for(var i = 0; i<6; i++) { web3.eth.getStorageAt(contract.address, i, callbackFNConstructor(i)) }

Running the above code from web3.js will return us the first 6 storage variables defined in the target smart contract. Then, by accessing in the storage array, we can find the 6 storage variables:

storage

(6) ['0x0000000000000000000000000000000000000000000000000000000000000001', '0x0000000000000000000000000000000000000000000000000000000061e8582a', '0x00000000000000000000000000000000000000000000000000000000582aff0a', '0x6afa09c95f78aa0d4427b26b649a9e39764fb2e42aa80a1628d0af9465f2dfc6', '0x5be1ffbd26ef932af6988e85bfa3168a66be9f53ad7a4036c81ec5578a9e12a6', '0x3cb5971a348f1a4224f2175e48a929cc30b29ff44f5886d651e48fce1acdab53']

The first storage variable ends with "001" so we expect to be a Boolean variable. **Locked** is the only Boolean variable and obviously is the first initialized storage array data. Then. We have the variable flattening=10; and the variable denomination=255. We can find these in the third storage data. Due to storage optimizations from Ethereum, we can find it in the same variable.

In order to understand better the concept, storage optimization follows the following rules:

1. Each index set aside for a storage variable allowing 256bits of data, optimizing for both storage efficiency and access speed,

2. Variables are indexed in the order they are defined in the smart contract (**contract.abi**)
3. When a variable needs less than 256bits to be represented, leftover space will be shared with subsequent variables if they fit. Otherwise, they will be in the next storage variable.
4. Array data always have to start from a new slot, thus it would start from slot 3. Since it is a bytes32 array each value takes 23bytes. If we consider that the value starts at index 0 stored in slot 3, index 1 stored in slot 4 and index 2 in slot 5.

Additionally, it is important to note that constants do not use this type of storage. Instead, all storage variables are managed as byte32 array data. There, we can find the **unlock()** function which requires the key to be a byte16 and then compares it and typecasts the data array[2] as a byte16 to achieve the comparison. As we understand, in order to unlock the vault, we have simply to take the first half of the bytes32 data to create a byte16 type data and then calling the unlock method with the byte16 we have created as an input.

By analyzing and combing all of the above information, we can understand that in practice the key is stored in slot 5(index/ 3rd entry).

We can read it by creating a variable from terminal and taking only this information:

key = await web3.eth.getStorageAt(contract.address, 5)

And it returns:

0x5dd89f7b81030395311dd63330c747fe293140d92dbe7eee1df2a8c233ef8d6d

This key is 32 byte however the requirements is to check in **unlock** and then it converts it into data[2] which is 32byte value to a byte16 before matching.

So we have to keep only the first 16 bytes. We can achieve this by running:

key = key.slice(0, 34)

Then, we can unlock the valt by using the command:

await contract.unlock(key)

And we have successfully completed this challenge.

5.8.1. Recovery

In this smart contract called **Recovery.sol** from Ethernaut, we have a smart contract that can create a very simple token by its own. Anyone can create a new token with ease by utilizing this smart contract. The goal on this smart contract is to find the lost smart contract address and then withdraw all of its ether.

We can see that this smart contract has a destroy function with a **selfdestruct()** operator on it. However, the **destroy()** function is not public payable, which means that we can't simply make a transaction and call it. We should find the smart contract's address and then call it by using the smart contract's address. As we know, smart contracts' addresses are deterministically calculated. From Ethereum's yellow paper we can read: "**The address of the new account is defined as being the rightmost 160 bits of the Keccak hash of the RLP encoding of the structure containing only the sender and the account nonce. Thus we define the resultant address for the new account**".

So, we can represent this function as:

address = rightmost_20_bytes(keccak(RLP(sender address, nonce)))

where sender address is the smart contract or wallet address that created this new smart contract. So practically, we can very easily type **contract.address** by console and find it.

Nonce is the number of transactions sent from the sender address or, if the sender is another smart contract, the nonce is the number of smart contract creations made from this account.

RLP is an encoder for data structure and is the default to serialize objects in Ethereum. The RLP encoding of a 20-byte address is 0xd6,0x94. Furthermore, for all integer less than 0x7f, their encoding it's just its own byte value. As a result, the RLP of 1 is 0x01. We should keep this for the next steps.

Keccak is the cryptographic primitive that compute the Ethereum SHA3 hash on any input. It is also referred as Keccak-256.

Now, we are ready to recalculate the address of the new smart contract, using an existing smart contract located at **contract.address**. It should be mentioned that nonce 0 is always the event of a smart contract's creation. Therefore, we will begin counting from 1. We know that RLP of 1 is 0x01 so by using Web3.js again, we will attempt to use the function described above:

web3.utils.soliditySha3("0xd6","0x94","0→xEa90EAAAB78241373b94C98a788524582DA2 e593" ,→ "0x01")

This command will return the hexadecimal value of:

0xa1ed98d854d1b4ee7a4aa44816a095b1578566ec7fd2162

46a8c0a6322f67331

We will keep only the last 40 characters:

16a095b1578566ec7fd216246a8c0a6322f67331

And we have found the smart contract's address. The only thing that is missing is the 0x at the beginning, ending up with the address of **0x16a095b1578566ec7fd216246a8c0a6322f67331.** We can check the address on Blockchain Explorer such as Etherscan.io, to see if holds 0.01 ether. Now that we have the address, we can attempt to call the **selfdestruct()** function. Therefore, we will encode a function call from console:

data = web3.eth.abi.encodeFunctionCall({

name: 'destroy' , type: 'function', inputs: [{

type: 'address', name: '_to' }] }, [player]);

And then we will create a transaction with our own player to sent the smart's contract ether to us when we trigger the **selfdestruct()** operation:

await web3.eth.sendTransaction({

to: "0x16a095B1578566ec7fd216246A8C0A6322F67331",

from: player,

data: data })

From this smart contract, we learnt that we could send ether to a deterministic address, even when the smart contract does not exist, at least for now.

5.9. Error Handling

For Explaining error handling, we will use a smart contract game from Ethernaut called **King.sol**. This type of scam has happened many times in the Blockchain world. These types of smart contracts are classic examples of Ponzi schemes. Whoever sends an amount of Ether that is greater than the current prize in the smart contract, becomes the new king. On such an event, the overthrown king gets paid the new prize making a bit of ether in the process.

It should be mentioned that when we click the submit button on Ethernaut to finalize the King's challenge, the Ethernaut attempts to call the fallback function to regain kingship. So, in order to guarantee that we will remain the king of the smart contract, we have to find a way to force all Ethernaut's transaction on the smart contract to fail. Auditing the code of the fallback function reveals that the external payable function has as a requirement that the sender must send more ether than the prize or the **msg.sender** be the owner. We should consider that all sent Ether is transferred to king and we become the king. Our value of ether is then set as the current prize, which an another user has to exceed to claim new kingship. The key of this challenge is the **king.transfer()** method which we can manipulate and force it to fail if the current king is a malicious smart contract which refuses to accept the transfer. In the Ethereum Based Blockchain, there are many ways for a transaction to fail but most of the time, we will primarily focus on these 3 options:

- **Out of Gas Error:** This error is one of the most common errors that can appeared on Ethereum based blockchains and it can happen when the receiving smart contract has a malicious payable function which consumes a large amount of gas in order to make the transaction fail as it over-consumes the gas limit. However, on Kings smart contract cannot be applied as we cannot make an internal call here,
- **Arbitrary Error** from the FallBack Function: With this approach, we are trying to execute a transaction calling a smart contract function designed to always fail,
- **Creating Transactions with smart contracts without a fallback function or a non-payable fallback function:** As demonstrated in previous examples, it is possible to initiate transactions and attempt to send Ether to smart contracts which lack such functions. When we are creating a transaction to a smart contract without a fallback function, it will fail. It will also fail if we try to send some ether in a function without a payable modifier in the fallback.

So, we can create a smart contract with a function which will send some ether on the Kings smart contract, but without a fallback function. Since the smart contract will not have a fallback function, the king smart contract cannot send money back. So, it remains stuck at we become the King forever. In order to achieve this, we create a malicious smart contract called **AttackTheKing.sol**

We can deploy this smart contract by using Remix IDE and during the deployment process, we have to deploy it with at least 1 Ether on it. In this scenario, it would likely be easier to deploy **King.sol** in Remix JavaScript VM in order to have virtual ETH minted in order to execute this example, as it needs a lot of Ether which potentially it is a bit difficult, to get in Ethereum Testnets.

The malicious smart contract offers a payable external fallback function, however it has a requirement in order to be executed, thus it will fail as it will can't handle the 1 ether that we sent it from the malicious smart contract and as a result we became a king forever.

5.10. Gas Fees Manipulation

On the previous example, we discuss the potentiality for an **"Out of Gas"** error as an attack vector on **Kings.sol**. Gas measures the amount of computational effort that is required to execute operations in Ethereum Blockchains as each operation requires a certain amount of gas in order to be executed.

At first, every transaction on Ethereum, including smart contract interactions, must specify a gas limit. This is the maximum amount of gas a user is willing to spend on the transaction. Setting an appropriate gas limit is crucial as if it is set too low, there is a potential for a transaction to fail due to insufficient gas ("**Out of Gas**" error). Reversely, if the gas prices are set too high, then the user risks overpaying for simple operations.

Besides gas limits, users must specify a gas price, which indicates the amount of Ether they are willing to pay per gas unit. Validators prioritize transactions with higher gas prices, affecting transaction speed. During periods of network congestion, a higher gas price is necessary to ensure timely transaction processing.

As transactions can fail due to **Out Of Gas** errors, there are potential attacks that could be based on this scenario. For instance, an attacking smart contract would try to consume all gas fees in a transaction in order to manipulate the smart contract to prevent the execution of specific tasks such as upgrading the state of Blockchain thus failing to record what happened on an attack, or even apply Denial Of Service Attacks (DoS Attacks) by sending transactions that consume all gas, causing legitimate transactions to fail. Finally, fallback functions, as we have in the King challenge, smart contracts without fallback functions or with non-payable fallback functions that receive Ether can cause transactions to fail.

There are multiple ways to protect the smart contract's execution from failing. For instance, by initializing the data structures to be as fixed sized (**byte32** over **string** for fixed-sized data, fixed arrays instead of dynamic) or avoiding the smart contract to call other external smart contracts. In case of conditional statements, the best practices shows that it is more efficient to use in the left the expressions that are most likely to fail, as Solidity starts to execute commands from left to right. If a statement fails, then it stops from executing the other ones, saving gas fees.

Development environments such as Truffle, Remix IDE or Hardhat, are offering tools to help the developers understand how gas costs during the development phase.

The Awesome Solidity Gas-Optimizations repository on GitHub offers a long set of papers, articles and other resources to make the developers further understand how to effectively gas fees management.

In the next examples, we will explore smart contracts with a focus on Decentralized Finance (Defi), designed to show attacks based on gas limitation attacks or to create infinitive loops in order to enforce a smart contract to fail.

5.11. Reentrancy Attack

One of the most critical attacks that can occur on smart contracts operating on Ethereum Based Blockchains is the Reentrancy Attack. This type of attack back in 2016 was the primary reason that the Ethereum Network split into two Blockchains: Ethereum and Ethereum Classic. A smart contract project called DAO (Decentralized Autonomous Organization) holding 150.00.000 dollars' worth of Ether, has been attacked and exploited, stealing over 50.000.000$ worth of Ether. The community was shocked back then as it was a huge fund loss, in terms of the times. As a result, the community started to try to identify way to revert the DAO hack, ending up with the community being divided into 2 main ideas:

- Reverting back to a previous state of Blockchain and revert the hack, saving users funds. However, this process would violate the model of trust of immutability that the Blockchain preservers. Each change in the state of Blockchain must be permanent and irreversible.
- Keeping the Blockchain as it was in order to promote immutability and trust in the decentralized world, however ending up with ton of assets stolen and lost, triggering panic sells or fear in the Ethereum world, making several people staying away from this technology.

The result was a hard fork of Ethereum Blockchain, ending up with 2 different chains (Ethereum and Ethereum Classic).

However, in order to understand how an attack like this happened in first place and how we can protect our smart contracts from vulnerabilities like this, we will present a vulnerable smart contract and show step by step how a Reentrancy attack can occur.

In order to be described more easily the process, we will represent the steps of a Reentrancy Attack as bullets:

1. **The initial call:** An attacker initiates a transaction to a vulnerable smart contract, usually one that handles funds such as a wallet or an exchange smart contract or other Defi related smart contract,
2. **Vulnerable Function Execution:** The smart contract processes the transaction as a part of its operations, transferring funds to the attacker's address. This is often executed through a function which includes external calls to other smart contracts, for instance, for sending Ether (**msg.value**),
3. **The Recursive Reentry:** Before the initial transaction is completed and the smart contract's state is updated (e.g. updating the balance of the funds), the attacker's attacking smart contract automatically calls back into the same vulnerable function of the original smart contract, practically creating a loop (a recursive reentry), without in practice let the smart contract to finish all of its operations and update the current balance,
4. **The Repeated exploitation:** Since the smart contract's balance variables haven't been updated yet, the recursive call finds the smart contract thinking that it is still in the original state, allowing the attacker in practice to withdraw funds again and again. The attack persists with repeated function calls, draining all the funds from the smart contract,
5. **Finishing the attack:** The attack continues calling the function again and again until the smart contract's funds are being eliminated or until an operational limit is reached, such as gas exhaustion.

From the above explanation, we understand that the key to a successful reentrancy attack lies in finding a way to prevent a smart contract from updating its internal state before making external calls, allowing the attacker to reenter the smart contract's function and exploit it repeatedly.

The Vulnerable Bank Reentrancy Attack

In this challenge we have a smart contract called **VulnerableBank.sol**, which is developed to accept deposits and allow withdrawals of Ether.

However, this smart contract is vulnerable to Reentrancy attacks.

The **withdraw()** function is not developed with the best security practices, as it first sends Ether to the caller, before deducting the amount from the user's balance. As a result, this can be exploited in order to execute a Reentrancy Attack.

An attacker can develop and deploy a malicious smart contract, designed to exploit VulnerableBank during its execution. So we have developed VulnerableBankExploit.sol.

The methodology is to first deposit a certain amount of Ether into VulnerableBank through the malicious smart contract in order to establish a balance and then the attack will initiate a withdrawal from the VulnerableBank through the malicious smart contract. As we can see in the code from the smart contract, we need to send more than 1 ether to achieve this. Upon calling the withdraw function, VulnerableBank transfers Ether to the malicious smart contract before deducting the amount from the attackers' balance within VulnerableBank.

The malicious smart contract receives the Ether and through its fallback function, automatically calls back into the VulnerableBank's withdraw function before the initial withdrawal transactions is completed.

As VulnerableBank has not yet updated its internal variables, having the attacker's balance, the smart contract believes that the attacker still has its original balance. As we understand, this critical security bug allows the withdrawal process to be repeated until draining all Ether from the smart contract.

To prevent such attacks, developers should ensure that all state changes precede external calls. In the case of the VulnerableBank, the balance deduction should occur before the Ether is sent out from the smart contract. Additionally, reentrancy guards such as OpenZeppelin's ReentrancyGuard.sol or using the transfer method (which limits gas sent to calls, mitigating reentrancy risk) can further secure smart contracts against such vulnerabilities.

So, to sum up, in order to defend our smart contracts against Reentrancy attacks, it is crucial to ensure that all state changes occur before external calls.

5.12. ERC20 Tokens

5.12.1. Implementing a Very Basic Token

We have developed a smart contract called **UNIToken.sol**.

In this challenge, we have to find a way to mint more UNITokens than these that already exist. After we are auditing the code, we can see, the code is very simple and can very easily be audited. A constructor sets the smart contract deployer as the owner, which variable is set as private, and mints the specified number of UNITokens. We can also find 3 functions, **transfer()** which as the name suggests, transfers UNItokens from the caller to a specific address which is being asked as an input and the number of UNItokens that the user wants to send, a function called mint which can be called only from the owner of this smart contract and mints new UNItokens and a function calls **isOwner()** which returns if the caller is the owner of the smart contract. However we can clearly notice that **isOwner()** is not a public view function and potentially the developer forgot to add the view in function **isOwner() public view returns(bool)** and instead he has written a function **isOwner()** which is **public returns(bool).** Which means that this function is not just a viewable function that returns if the owner is the **msg.sender** but it can clearly be executed by everyone and set the **msg.sender** as the owner of the smart contract and then returns if the owner is the **msg.sender** which obviously will always return true.

And as we became the owner of the smart contract, we can pretty easily satisfy the requirement from the function **mint()** and print millions of **UNItokens** and thus hyperinflating the cryptocurrency.

This smart contract was almost developed perfectly, however 2 issues in one small function were enough to take the ownership and print a lot of tokens.

In order to reduce the probability of errors in development of our own tokens, we can use the ERC20 standard. We can easily create an ERC20 token by using Remix IDE with the plugin of Cookbook.dev, which includes an Audited version from OpenZeppelin for creating ERC20 based tokens.

For instance, the code that we have written before it would be simply this:

```
pragma solidity ^0.8.10;

import "simple-token/@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract UNItoken is ERC20 {

  constructor(

    string memory name,

    string memory symbol,

    uint256 totalSupply

  ) payable ERC20(name, symbol) {

    _mint(msg.sender, totalSupply);

  }

}
```

When we use smart contract declaration in Solidity like **contract UNItoken is ERC20**, it indicates that UniToken inherits the functions from a smart contract named ERC20. In the context of developing and especially in smart contracts, inheritance and interfaces are being used to extend the standard interfaces and to reuse code, ensuring that the code behaves in predictable ways across different application and wallets. That also means that smart contracts can be developed as the "basis" and then be audited by reputable companies to identify and patch potential vulnerabilities. Then, they can be used and extended by other developers.

Based on this, it's important to highlight some key characteristics of interfaces:

- In practice, interfaces serve as a smart contract template, defining a set of functions without implementing them, allowing different smart contracts to interact with each other regardless of their implementation,
- Interfaces cannot be inherited from other smart contracts, but they can inherit from other interfaces,
- All declared functions in interfaces must be external so they could be called from the other smart contracts,
- They cannot declare a constructor, state variables or modifiers,
- Smart contracts can inherit interfaces as they would inherit other smart contracts.

5.12.2. Time Locked tokens

In this challenge we have an ERC-20 token called the Naught Coin. Naught Coin is an ERC20 token and we are already holding all of them. The catch is that these tokens are locked by their code, preventing any transfer for a period of 10 years.

Such lock mechanisms are very common in ERC20 tokens deployed across various Blockchains. Most of the times, projects with locking periods turns out to be scams pretending to be real tokens for creating transactions and exchanging however often their developers are hiding from the community that their tokens are programmed to stay locked for a predefined period of time, to contribute to pump and dump schemes.

On this challenge we are requested to find a way to bypass the lock and then move the Naught Coins to another address. By auditing the smart contract's code, we understand that it inherits functions from ERC20 tokens to be functional.

Understanding the ERC20 standards thoroughly it is of high importance as all these kinds of attacks are based on this. Also, we should consider that most of the times, especially in scams, the developers often poorly develop their smart contracts to save time and money, so it is very likely that they have made some common issues implementing standards by copy pasting for well known GitHub repositories or using AI tools such as ChatGPT to develop their own tokens.

On Ethereum's Repository on GitHub, we can find more information about how ERC20 tokens are working. https://github.com/ethereum/ercs/blob/master/ERCS/erc-20.md

After studying the abstract, we can understand that a token in order to be ERC20 compliant with its specification, it must have implemented the following functions as an interface:

contract ERC20Interface {

function totalSupply() public constant returns (uint);

function balanceOf(address tokenOwner) public constant returns (uint balance);

function allowance(address tokenOwner, address spender) public constant returns (uint remaining);

function transfer(address to, uint tokens) public returns (bool success);

function approve(address spender, uint tokens) public returns (bool success);

function transferFrom(address from, address to, uint tokens) public returns (bool success);

event Transfer(address indexed from, address indexed to, uint tokens);

event Approval(address indexed tokenOwner, address indexed spender, uint tokens);

}

After studying these functions from the documentation and reviewing the Naught Coin's smart contract, we notice that **transferFrom()** function is being modified to add **lockTokens()** modifier, however NaughtCoin did not implement the **tranferFrom()** function. That means that the StandardToken implementation of **transferFrom()** is being called thus we can bypass the token locking mechanism by calling this function and withdraw the Naught Tokens.

So, we see our balance of NaughtCoins by typing:

(await contract.balanceOf(player)).toString()

It returns 1000000000000000000000000 NaughtsCoins. These NaughtsCoins have to be sent to another address. So, we approve our address in order to be able to send a predefined number of NaughtCoins:

await contract.approve(player, "1000000000000000000000000")

And then we are ready to send all these NaughtCoins to a random Ethereum wallet:

await contract.transferFrom(player,"0xDA0bab807633f07f013f94DD0E6A4F96F8742B53", "1000000000000000000000000")

Now we run again:

(await contract.balanceOf(player)).toString()

And it returns 0, so we have successfully completed this challenge.

5.13. Overflow Underflow Attacks on Tokens

Arithmetic overflows and underflows are another common vulnerability in smart contracts written in Solidity.

At first, a uint variable in reality means unsigned integer of 256 bits. These variables can store zero or positive number values, but they can never be negative. In practice these variables can get any range of values from 0 to 2^256 -1. The overflow attack exploits these variables in situations where a value higher than 2^256 is attempted to be stored in a uint variable. Since this value cannot be stored in the variable, it results in an overflow, resetting the value to zero. This can be easily explained with a uint8 variable, for example if we have 11111111 (in binary) and try to add one more, this will end up in overflow to 00000000, overflowing the result to zero.

Conversely, in an underflow attack, if we enter an integer number which is lower than the supported one in order to drive to -1 which cannot be happen due to unsigned uint number range and thus going to the maximum value which in a similar case with the previous case, it would be 1111111.

Solidity versions prior to 0.8.0 did not provide support in error handling errors for overflows and underflows. As a consequence, developers must be very curious when they develop smart contracts that involve numeric operations.

To further explain these attacks, we will use the Ethernaut's smart contract called Token.sol.

By auditing the code, we find a mapping **balances**, which can store unsigned integer256 values.

Next, we can see an uint public variable called **totalSupply**.

In the constructor we can see that is being initialized under the deployment of the smart contract.

Furthermore, we can see a viewable function named **balanceOf()** which takes an address as input and returns the balance of this address of these tokens. To check the token balance of our address, we can run:

await contract.balanceOf(player)

And we can see that we have 20 tokens.

The **transfer()** function takes an address **to** and an uint **value**, and includes a requirement that the user's balance minus the value be greater than zero, so it would be always true as the variable is defined as uint, thus it will always be greater or equal to zero.

In the next line, we see that it substitutes the value from the user's balance. This line can be vulnerable to underflow attacks. So, let's attempt to transfer our user more tokens than intended.

We copy a random Ethereum address to use for the **_to :**

contract.transfer('0x5C4A7D0f061B357809eD7cB6A7cD16fa5C3d1e74', 20+1)

Since 20 - 21 will result in an underflow.

Now let's run the **balanceOf()** again to see our tokens:

await contract.balanceOf(player)

and we can see that due to underflow, that everyone owns 67108863 tokens.

To mitigate overflows and underflows, developers can use OpenZeppelin's SafeMath library for arithmetic operations within their smart contracts, providing an additional layer of security by automatically checking for these vulnerabilities.

5.14. Short Address Attack

Short address attacks target smart contracts by manipulating transactions with addresses shorter than the expected 20 bytes (e.g. 18bytes), causing unintended behavior in the smart contract's execution into reading more data than was sent. The vulnerability happens when smart contracts fail to properly validate the length of an address input, assuming addresses less than 20 bytes are valid due to incorrect checks in the length. An attacker can call a vulnerable function with a short address, deceiving the smart contract into recognizing it as valid due to flawed length verification, potentially leading to failed transactions or incorrect value transfers. Ethereum addresses are typically 20 bytes long, however if an attacker uses an address shorter than 20 bytes, the Solidity smart contract will automatically pad the difference on the right side with zeros to ensure that the address is of appropriate length.

In order to explain the attack better, we can suppose that we have a user A who owns 100 tokens and wants to transfer 10 tokens to User B using a platform which lacks Input Validation. For simplicity, we will assume that the Ethereum format is 4 bytes. User B's address is supposed to be 0xdddddd00 but user A either by mistake or intentionally, enters 0xdddddd, without the last two zeros (00). The smart contract reads it as a valid input address, converting it to 0x00dddddd for the transaction. Then, it takes the token amount (10 or 0x0000000a) along with a hypothetical function signature (0xaabbccdd), to create the transaction: 0xaabbccdddddddd0000000a. However, dissecting this into a 4-byte signature and 2 4-byte words, we end up with: ['0xaabbccdd', '0xdddddd00', '0x00000a??']. The Ethereum Virtual Machine pads missing input data with zeros, ending up with ['0xaabbccdd', '0xdddddd00', '0x00000a00']. 0x00000a00 in hexadecimal, which is 2560 tokens, not the 10 tokens intended by user A.

This type of vulnerability is particularly risky for large, communal wallets, for example those who are being used by exchanges, where the total balance represents the cumulative assets of many users. If such a wallet had over 10.000 tokens, exploiting a short address could enable an attacker to withdraw significantly more than their own balance, depending on the number of trailing 0s in the manipulated address. In most reputable development environments such as Remix IDE, static analysis tools automatically identify issues like this and shows error messages in the compilation or if they are tried to be sent as an input, they mark the transaction as incomplete, in order to avoid input validation issues like this. However, it's critical for developers to understand these vulnerabilities and implement appropriate checks, especially for smart contracts that handle significant value or are part of a critical infrastructure.

Chapter 6: Topics In Decentralized Finance

In previous examples related to common vulnerabilities in smart contracts, we showcased various smart contracts designed for solving general issues, including random number generation and some exploits associated with ERC20 tokens. While many of these many smart contracts are simplified examples derived from widely used ones, there are still smart contracts that interact with a predefined number of other smart contracts to serve the users of these smart contracts.

In many complex projects, in order to be functional, a set of multiple smart contracts running on a Blockchain are being used and they are often developed to communicate with the real world and internet or even with other Blockchains, associating assets across to multiple chains.

Decentralized Finance, or simply named as DeFi, represents a radical change in the financial sector, moving away from the traditional, centralized institutions towards a more open and accessible financial ecosystem powered by blockchain technology with blockchain networks communicating each other and connecting with the real world. This innovative model empowers users by democratizing access to financial services to even the countries with underdeveloped financial sector and ensuring that they are not just for the privileged of a few but for anyone with an internet connection. At the heart of DeFi are smart contracts which in practice are also self-executing smart contracts with the terms of the agreement directly written into code. These smart contracts combined, are building DeFi applications, building a Defi world, enabling automated, secure, and transparent financial transactions. Blockchain technology's inherent transparency plays a pivotal role in building trust within the DeFi ecosystem. Users can audit smart contracts and transactions openly, fostering a trustless environment where transactions and smart contract functionalities are verifiable by anyone, enhancing the system's credibility and reliability.

The complexity of financial transactions in the DeFi space demands innovative and thoughtful design of smart contracts in order to remain transparent and secure, as changing smart contracts cannot be that easy as changing applications as it happens in the conventional systems. These smart contracts must not only be secure but also efficient and flexible, capable of adapting to the dynamic needs of the DeFi market and its participants.

The radical changes that have happened due to Bitcoin, Ethereum and DeFi applications comes with significant security imperatives, given the vast sums of digital assets managed by these smart contracts. As we can understand, the security of these smart contracts is of high importance, as vulnerabilities could lead to substantial financial losses and losing trust in the decentralized world. Therefore, developing and maintaining secure security protocols is non-negotiable, ensuring that assets are safeguarded against attacks and technical failures or even being secured from external manipulations. The most well know examples of Defi products are Decentralized exchanges or lending protocols as they are enabling trading without the need for centralized intermediaries, which are legally required to maintain records of user's information through Know Your Customer (KYC) processes, to prevent money laundering and financing of illegal activities. However, these methods compromise user's privacy.

Decentralized Exchanges and lending protocols are enabling users to lend, borrow and even engage in collateralization and interest rate determinization (for instance, staking in Defi apps or validating pools) and liquidation by using algorithms or data exported from other protocols. Decentralized Finance platforms can be described as a set of multiple smart contracts deployed to multiple blockchain networks in order to provide the abilities that have been described before. The next subchapters will discuss several concepts that enable a collection of smart contracts to be functional as decentralized financial systems and the security risks associated with these technologies, including:

- **Oracles:** Technologies to communicate with the Real World,
- **Stable coins:** Methodologies to represent real world assets in the Digital World,
- **Atomic Swaps:** Technologies to support direct cryptocurrency exchanges between different blockchains without the need for intermediaries,
- **Bridges:** Technologies to move assets across multiple Blockchains,
- **Non Fungible Tokens:** The tokenization of digital assets,
- Security improving technologies to secure these assets in terms of Access Control and upgradability.

6.1. Non-Fungible Tokens (NFTs)

Having discussed about Blockchain's main cryptocurrencies, tokens and especially ERC-20 tokens standard, now we have to discuss about Non Fungible Tokens or simply names as NFTs.

NFTs have gained tremendous popularity by mid-2021 notably with collections such as the Bored Ape Yacht Club, featuring animated monkeys in various scenarios and has sold for millions of US Dollars. These tokens, often featuring animated monkeys in a variety of situations, represent just the tip of the iceberg in terms of the potential of NFTs. Initially, NFTs are tokens represent assets from both the digital and physical worlds, including university degrees, certifications, special in-game items, or even represent real state within the

Blockchain ecosystem. Unlike main Blockchain cryptocurrencies or ERC-20 tokens which are fungible and identical for exchange purposes, NFTs are unique and distinct. That means that we can mint a specific NFT and assigned it to a specific address and it cannot even be transferred if it is programmed in that way. This process can be beneficial for instance in cases of minting tokens representing a university degree, which typically should not be transferred to other users. Furthermore, in case for real state of ownership, we can issue a NFT to 2 users, representing equal 50%-50% ownership. Each NFT has a unique identifier that sets it apart from others, similar to how a rare trading card might differ from common ones, making them identical for videos, movies, gaming or anything that we want to prove ownership and authenticity of digital goods. This process not only proofs the ownership of digital assets, but also supports the creators or developers to monetize their creations directly and sustainably.

NFTs in the Blockchain world comes with 2 different standards:

- ERC-721
- ERC-1155

The ERC-721 standard serves as the standard for NFTs creation as it satisfies all of the characteristics that have been discussed before. It is the most famous standard in the NFT world allowing the creation and transfer of unique NFTs and it can serve as the proof of ownership.

Conversely, ERC-1155 represents an improved version of initial NFT standard, introducing multi-token standard that enables a single contract to represent both fungible tokens and non-fungible tokens (NFTs). ERC-1155 paves the way for more efficient transactions by enabling the transfer of multiple types of tokens at once, reducing the complexity and costs of transferring tokens on the Ethereum network.

ERC-1155 supports a wide range of digital assets and allowing for the creation of both unique items and items that exist in multiple copies, making it ideal for gaming, art, and other digital collectibles. Furthermore, ERC-1155 improves efficiency through supporting batch transfers of multiple token types, reducing transaction costs and complexity compared to ERC-721, which generally handles only one token type per transaction.

6.2. Flashloans

Flashloans are an innovative financial instrument available in Decentralized Finance across Blockchain Networks. Flashloans offer an uncollateralized loan option in which users are able to borrow any available amount of assets from a liquidity pool without providing any security or upfront collateral if the loan can be borrowed and repaid within the same transaction block.

Flashloans can be beneficial for instance as users could find a Decentralized Exchange A, offering for instance Ether for a price of 2000euro and Decentralized Exchange B buys ETH for a price of 2200euro. That means that a user could take 100ETH from the flash loan to buy ETH on DEX A. With a potential fee at rate of 0.3%, the total repayment amount could be 100.3 ETH. Immediately after, within the same transaction, the user sells the 100ETH on DEX B for 305.000euro, that means the user earned 2000euro without initially owning those ETH. However, it is important to note that executing a flashloan like this, individuals must possess enough Ether to cover the transaction fees. Nonetheless, the cost is significantly smaller than the amount that can be borrowed through the flash loan.

This short example explains how a user can successfully execute a strategy using flashloans without having capital, to profit from the price differences between the two decentralized exchanges. However as we understand, flashloans can be beneficial for opportunities for capitalization, its crucial to acknowledge they carry several risks and as most of applications are susceptible to exploitation. In the next challenges, we will show attacks associated with flashloans.

For our flashloan smart contract examples, we will use smart contracts developed by Damn Vulnerable Defi. Damn Vulnerable Defi as the creator explains, serves as a for learning offensive security of Defi smart contracts on Ethereum and Solidity. Unlike OpenZeppelin's Ethernaut, which uses its own infrastructure which automatically deploys smart contracts in Sepolia Ethereum Testnet, allows for interaction via a browser with web3.js and Metamask.

6.3. Denial Of Service attack in flashloans

In the challenge, called Unstoppable.sol from Damn Vulnerable Defi, we are requested to perform a denial of service attack in the smart contract, to make the vault stop offering flash loans. These smart contracts is working by using an ERC20 token called Damn Vulnerable Token(DVT).

At first, we audit the smart contract's code to understand better how this flash loan operates. The ReceiverUnstoppable.sol smart contract is designed to use the interface IERC3156FlashBorrower protocol, allowing it to issue flash loans transactions and it is used by the UnstoppableVault smart contract. To support these operations, the smart contract uses 2 functions, **onFlashLoan()** which is being called by the **UnstoppableVault** smart contract when a flash loan is being created. In order to create the flashloan, it checks the initiator, the token and the fee if they are valid and then, it approves the flashloan to be able to be executed. Furthermore, we have the **executeFlashLoan()** which calls the **flashLoan()** function by the **UnstoppableVault** in order to create a flashloan.

Then, we audit the UnstoppableVault.sol smart contract which implements the IERC3156FlashLender and ERC4626.sol interface and enables the flashloan initialization and offers several functions. For instance, **maxFlashLoan()** returns the maximum amount of tokens that can be borrowed from a flashloan. On the other hands. **FlashFee()** calculates the flash loan fee and returns its value, **setFeeRecipeint()** which update the address of the fee recipient and **totalAssets()** which returns the total amount of tokens held by the smart contract.

Finally, the is the **flashLoan()** function which creates the flash loan and triggers the fallback function of the receiver, which is the function that stores the vulnerability of this Defi system.

We check its code and we can see that includes:

uint256 balanceBefore = totalAssets();

if (convertToShares(totalSupply) != balanceBefore) revert InvalidBalance();

which was developed in a way to enforce the ERC4626 requirements which are a standard proposed for tokenized vault in Defi applications in order to create a standardized approach for handling deposits and handles within a vault in order to determine the rewards of staked tokens.

We have the assets which are representing as DVT that users deposit and withdraw from the vault and shares which are the vault tokens, minted or burnt for uses based on the proportion of their deposited assets. **convertToShares()** function from ERC4626, calculated the number of shares the vault should mint, based on user's deposited assets. The **!=balanceBefore** means that **totalSupply** of vault tokens to always be the exactly the same amount with **totalAssets** before executing **flashLoan()** function. If it reverts, the smart contract becomes inactive.

However, the **totalAssets()** function has been overridden to return the balance of smart contract's vault asset **((asset.balanceOf(address(this)))**. We understand that the smart contract introduces an alternative system which relies on monitoring the supply of vault tokens.

To exploit this, all we have to do is to create a conflict between the 2 accounting systems by transferring DVTs directly to the vault. This will end up disrupting the balance, making the **(convertToShares(totalSupply) != balanceBefore)** to fail, deactivating the flash loan.

As the flashLoan() function is deactivated due to the bug related on the accounting systems, the transaction will always be reverted without depending on user's input.

In order to achieve this, we can simply send DVT directly to the pool:

await token.connect(player).transfer(vault.address, 1)

And we have successfully completed this challenge.

6.4. Broken Access Control in Flashloans

On this challenge, we will use an another Defi application from Damn Vulnerable Defi, called Naive Receiver.

This Defi application has a pool with 1000ETH in its balance offering flash loans, with a fixed fee of 1ETH per transaction. A user has deployed a smart contract with 10ETH on its balance which is capable of interacting with the pool and receiving flash loans on ETH. The purpose of this challenge is to take all ETH out from the user's smart contract (not the flashloan pool) with only one transaction.

Naïve receiver as the description of the challenge suggests, it is consisted of 2 smart contracts:

- NaiveReceiverLenderPool.sol, which is the flashloan pool smart contract,
- FlashLoanReceiver.sol, which is the smart contract deployed by the user.

By auditing the code of NaiveReceiverLenderPool.sol, we find out that has multiple functions for different actions. For instance, the **receiver()** function receives the amount of requested loan (and uses the IERC3156FlashBorrower interface), the token which borrows (ETH), the amount of tokens which will be sent to the receiver smart contract and a data variable for optional data which might be needed.

So, as we understand, the receiver on his smart contract, before auditing his own smart contract, we expect to include the interface mentioned before and **onFlashLoan()** function which in order to be functional has to return the constant CALLBACK_SUCCESS in case that the smart contract executes all of the actions successfully.

Moreover, the smart contract follows best practices on this scenario and validates the balance exceeds the original balance with an extra 1ETH on it from the transaction fees that are enforced in user.

Now that we have understand how **NaiveReceiverLenderPool.sol** works, we can audit **FlashLoanReceiver.sol** to identify any bad implementation from the user that has taken a flash loan, that could be exploitable.

We are reading the smart contract's code, and in the first lines we can figure out a critical flaw in the function **onFlashLoan().** While the function implements many of variables (address token, uint amount, uint fee etc.) we can clearly see an address which is not initiate any variable. As a result, it could be the **msg.sender** who initiated the flash loan (in this, the flashloan pool), with an extra of the other variables added from the user. So in practice, the smart contract is permitting everyone to take a flashloan by himself.

This can be achieved as an Access Control flaw and we could try to create a transaction with 1ETH (as the flashloan in order to be executed needs 1ETH for fees) and then take a flashloan. However, the challenge requires from us to execute the attack only with one transaction. So, we create a smart contract called AttackUnstoppable.sol with the interface mentioned before which would use a for loop for 10 times which would create 10 flashloans.

6.5. Truster Lender Pool

In the challenge called Truster from the Damn Vulnerable Defi we have a new pool which is offering flash loans of DVT tokens for free. The pool owns 1 million of DVT tokens and we are requested to withdraw them. It mentioned that our wallet owns 0 DVT tokens.

We audited the code of TrusterLenderPool.sol and we can see that there is constructor initializing the smart contract with a reference to DamnVulnerableToken which is an ERC20 token (and an another function called **flashLoan()**) which allows a flash loan operation where a specified amount of tokens is loaned to a borrower who can execute arbitrary code via a call to the target address with the condition that the borrowed amount is returned by the end of the transaction.

The first idea which we could have tried is to try to execute Reentrancy Attack. However, on the code we can find out that this smart contract utilizes ReentrancyGuard.sol from OpenZeppelin, to be protected against Reentrancy Attacks.

So, let's go back to the function **flashLoan()**.

As we have read in the description it is not checking that the variable **borrowAmount** is set as 0, so in practice we cannot just steal the funds from the lending platform, as it would understand that we don't send back the funds we took in first place. At the same time, it does not check the address in the **borrower** or **target**. So as we can notice, we can create a normal transaction as a call target.call{value: value}(data), which enables the execution of the **flashLoan()** function by the lending smart contract, which it makes an external call to the target address with user-supplied data, opening a door for a potential misuse as we could set the address of the pool's own token an then invoke any method under the pool or its ERC20 token standards. DamnVulnerableToken as has been mentioned before, it is an ERC20 token as the tokens we have discussed in previous chapters. By utilizing the **approve(spender,amount)** function from the ERC20 standard, we will be able to permit a specified spender to withdraw tokens from it. At the same time, the attacker could claim the approved tokens using the tranferFrom() function which is also in the ERC20 standard.

So, we create a malicious smart contract which queries the total balance of tokens held by the TursterLenderPool and then it requests a flash loan from the pool with an amount of 0. Then, it exploits the flash loan's function call capability to execute an arbitrary command, in this case, the ERC20's tokens **approve()** function. As the **approve()** function is being executed in practice we approve ourselves to transfer the pool's entire balance. With the approval in place, it immediately transfers the total pool balance to the attacker (**msg.sender**), with the **tansferFrom()** function from ERC20 and we have successfully exploited this smart contract.

As a lesson from this challenge, we should consider of course not allowing flash loans with amount of 0 and we should also use in the smart contracts, Input Validation as we know from attacks from the normal infrastructure for example with SQL injection in the databased or URL filtering in the webservers, in order to avoid a malicious user or smart contract to execute action like this shown on this writeup.

6.6. Oracles

As we have clearly explained, smart contracts cannot by themselves have access to data outside of their own Blockchain network. For a smart contract to function smoothly and needs to process data from the outside world (i.e. outside of its Blockchain Network), it requires a form of "agreement", transferring digital data from the real world to Blockchain. These technologies are called oracles. Oracles are services that transmit and verify real world data and submit this information into smart contracts thus triggering changes in the state of blockchain. Oracles provide smart contracts with external information which can be programmed to trigger a predefined action to the smart contract. This external data typically originates either from applications with Big Data (such as healthcare, libraries etc.) or from hardware from Internet OF Things (IoT) devices, like temperature sensors or CO2 monitors etc.

Oracles can be categorized into four types:

- **Input Oracles**, which fetch data from the real world to the blockchain world, supplying smart contracts with the information needed for processing or triggering actions,
- **Output Oracles**, which retrieve data from the blockchain world and transmits them to the real world. The smart contracts are capable of triggering actions in the real world externally,
- **Cross-Chain Oracles**, offering interoperability within the blockchain ecosystem, bridging the communication gap between disparate blockchains, as blockchains cannot typically communicate with each other. Cross-chain Oracles also are acting as bridges triggering of actions from one blockchain to another.
- **Compute-enabled Oracles**, which are preserving a process of computing data off-chain in order to minimize costs associated with the blockchain's consensus mechanisms. Therefore, compute enabled Oracles offer users the opportunity to process data off-chain, bypassing any limitation a Blockchain network has, including legal issues, for instance with GDPR from EU when you want to anonymize data. Data like these cannot be stored on-chain, as there would be legal issues as these data could not be deleted after being processing as all information stored on a blockchain is permanent in its block history (as we have explained with the smart contract with **selfdestruct()**). Moreover, Compute-Enabled Oracles can generate true randomness

within the Blockchain ecosystem, addressing challenges like those identified in a previous example.

A primary challenge with Oracles is that people need to trust these external sources of information, whether transmitted by a person or a sensor.

Smart contracts are designed to enforce agreements through code and the consensus mechanisms of Blockchain Networks as smart contracts code usually is predefined and cannot be modified directly once deployed. If we use data outside from the Blockchain world that can modify or change the outcome from the execution of a smart contract, it can be expressed as undoing the real reason why we are using Blockchains and smart contracts at first.

In practice Oracles are third party services which are not integrated into the blockchain's consensus mechanism and thus they are not governed by the blockchain system's inherent mechanism. As we can understand, this setup could end up to man-in-the-middle attacks between the smart contracts and Oracles. Attackers could try to manipulate a smart contract which its functionality is known in order to trigger a predefined action (for example, sell X tokens in case the average price of Ethereum in centralized exchanges is at XX euro).

Oracles are divided to two main groups such as:

- **Centralized Oracles**, which have the main problem that has been described before,
- **Decentralized Oracles**, which aim to mitigate the central point of failure issue by distributing the data sourcing and verification process across multiple nodes.

6.6.1. Centralized Oracles

Centralized oracles operate under the control of a single entity or organization, serving as a bridge between blockchain smart contracts and external data sources. However, as it was described before, the primary concern with centralized oracles lies in the risk of a single point of failure.

If a centralized oracle is compromised, whether through security vulnerabilities being exploited, or from corruption in governance of an organization which possibly wants to manipulate an act or outcome, or due to frequent downtimes typical in the centralized world or even from DDoS attacks (Distributed Denial of Service attacks) which attackers are trying to make the centralized oracle unresponsive.

Any of these issues can severely damage the reliability and security of smart contracts dependent on a centralized oracle.

In case of a Defi Application, turning bitcoins into USDT, if it relies on an oracle to set Ethereum's marker price, an attacker could try to attack the infrastructure of the company to exploit potential vulnerabilities and then manipulate the API which checks Ethereum's price in the markets. On this potential attack, the attacker could easily alter the price of Ethereum maliciously to 0.01 USDT and then buy million worth of Ethers for some USDTs. Such a scenario could potentially be exploited to purchase NFTs or other assets at artificially reduced prices, as we will see in the next attacking scenario.

To reduce the risks associated with using single entities managing centralized oracles, the community has started to create and use Decentralized oracles.

6.6.2. Decentralized Oracle Networks

A Decentralized Oracle Network (or simply as DON) can be established by using various different decentralized nodes run by independent operators which are being used sourcing and authenticating data from diverse off-chain sources. Within a DON, data is collected by numerous sources to gather, analyze and aggregate results. As a result, DONs can enhance security compared to centralized oracles.

In Decentralized Finance for instance, a smart contract can request the real-time price of ETH to USDT from DONs, with enhanced security by aggregating data from various oracles, from multiple websites or centralized exchanges. Therefore, the smart contracts can place greater trust in a network with decentralized oracles instead of requesting the price from a specific centralized Oracle, which would be ask a specific website.

Chainlink represents the most famous example of a Decentralized Oracle Network and it is widely adopted in the Blockchain Community. Chainlink supports a lot of Blockchains such as Ethereum, Polygon, BNB chain, Avalanche, Optimism, Arbitrum and BASE.

Chainlink offers reliable Decentralized Oracles with extensive functionalities which heavily improve the way the Blockchain networks work and communicate each other and enhance the user's experience.

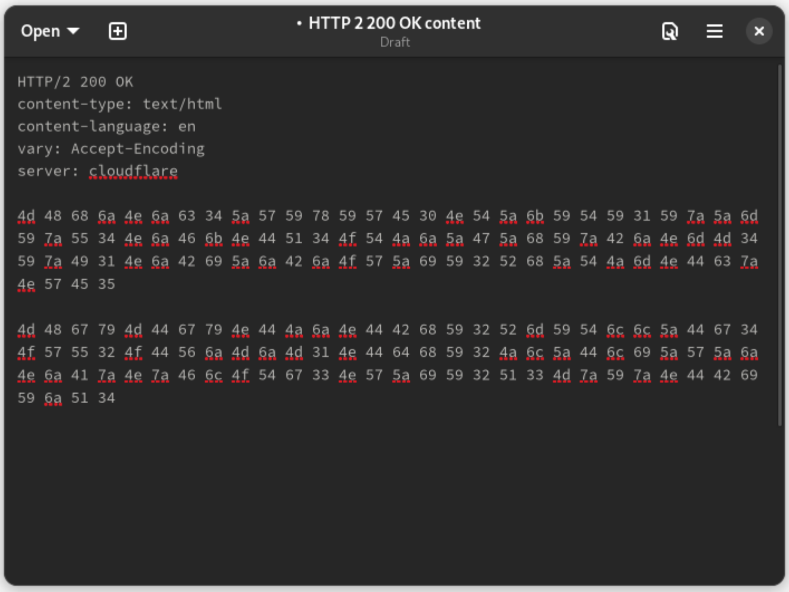
The most famous projects run by Chainlink are:

- **Chainlink VRF (Verifiable Random Function):** Chainlink VRF introduces randomness in the Blockchain world. As we have described before, Blockchains cannot produce true randomness by their own. For this reason, smart contracts can utilize Oracles such as Oracle VRF which can produce randomness and ensure that the process remains tamper-proof and verifiable by all participants, addressing a significant challenge in creating trustless systems,
- **Chainlink Price Feeds:** Chainlink Price Feeds provide decentralized, high-quality, and tamper-proof price reference data for a wide range of cryptocurrencies, fiat currencies and other assets in the Blockchain ecosystem. Practically, they solve the problems that have been discussed before and they can be very beneficial for Defi applications, providing them with accurate and up to date prices from the real world,
- **Chainlink Cross-Chain Interoperability Protocol (CCIP): CCIP** serves as a proposed standard for creating secure and interoperable messaging between different blockchains, enabling the transfer of tokens and data across multiple networks, foster greater connectivity across the entire blockchain ecosystem,
- **Chainlink Automatization (V2):** Chainlink Automatization offers consensus-driven automatizations for smart contracts, making them capable of to process data off-chain, significantly reducing the gas prices. Furthermore, it supports to setup triggers to launch automatic tasks on blockchain.
- **Chainlink Proof of Reserve:** Chainlink Proof Of Reserve provides smart contracts with the ability to automatically verify the collateralization of on-chain assets with off-chain accounts in real-time. This methodology is particularly important for stablecoins and other asset-backed tokens as they can ensure transparency and trust in their backing assets. In the following chapter, we will delve deeper into stablecoins and their security aspects.

6.6.3. NFT and Oracle price Manipulation based on Leaked private keys

In this challenge, we will explore the Defi application named Compromised which is a slightly different scenario from the previous shown ones, as it tries to present an entire company's infrastructure, managing smart contracts related on NFTs, Oracles and leaks from web servers and how the interaction between on-chain and off-chain data could be fatal.

The challenge begins with a response from Defi's application web server. While it is not directly related on the smart contracts, the response likely contains some valuable information.



```
HTTP/2 200 OK
content-type: text/html
content-language: en
vary: Accept-Encoding
server: cloudflare

4d 48 68 6a 4e 6a 63 34 5a 57 59 78 59 57 45 30 4e 54 5a 6b 59 54 59 31 59 7a 5a 6d
59 7a 55 34 4e 6a 46 6b 4e 44 51 34 4f 54 4a 6a 5a 47 5a 68 59 7a 42 6a 4e 6d 4d 34
59 7a 49 31 4e 6a 42 69 5a 6a 42 6a 4f 57 5a 69 59 32 52 68 5a 54 4a 6d 4e 44 63 7a
4e 57 45 35

4d 48 67 79 4d 44 67 79 4e 44 4a 6a 4e 44 42 68 59 32 52 6d 59 54 6c 6c 5a 44 67 34
4f 57 55 32 4f 44 56 6a 4d 6a 4d 31 4e 44 64 68 59 32 4a 6c 5a 44 6c 69 5a 57 5a 6a
4e 6a 41 7a 4e 7a 46 6c 4f 54 67 33 4e 57 5a 69 59 32 51 33 4d 7a 59 7a 4e 44 42 69
59 6a 51 34
```

A typical web server is a web2 style server utilizes HTTP (Hypertext Transfer Protocol) and additional other protocols to handle and respond to client requests made over the World Wide Web (www). The primary function of a web server is to store, process, and deliver web pages to users. This process involves listening for requests from web clients (typically browsers) and serving them with HTML pages, images, and other content. On this screenshot, we see the message sent from a web server which could be recorded for instance via a package sniffing tool.

The **HTTP/2 200 OK** segment of the message indicates the status from the HTTP response showing that it is using the http protocol and the status 200 OK indicates that the request has been successfully processed by the server and the browser (or client making the requires), should expect to receive a payload (which could be the page requested, data or some resource) in the response body.

The interesting part from this response are the hexadecimal strings, which may contain some useful information to use in the smart contracts that we are requested to audit.

We can try to convert one of the 2 hexadecimal to ascii via online tools or on GNU/Linux systems by running this command:

echo "the_hexadecimal_string_here" | xxd -r -p

which will produce the asci version of this string. As we check the result, we can see that it looks like the asci generated is encoded in base64. Base64 is an encoding scheme which converts binary data into an ASCII string, making it suitable for transmission of images, documents, and other binary data within HTML or for data transfer, over text-based protocols. The data in base64 are being encoded into 64 characters. However, the key here is that they are being encoded, not encrypted. That means that everyone with a base64 to string decoder tool could decode them and read the information stored on them. We can decode base64 again by using online decoders or by using GNU/Linux terminal by using the command: **echo " the_hexadecimal_string_here" | base64 –decode**

And we find out that in practice, the encoded base64 text was a private key for a wallet. Let's decode the other one too. We can do the above process into a single command by using:

echo "the_he " | xxd -r -p | base64 –decode

After this, we have ended up with 2 private keys which have been leaked in practice. Now, let's audit the Defi's smart contracts to see any pattern which having these private keys might be leaked.

In the description of the challenge after the web server response, we are notified that the related on-chain exchange is selling NFTs called "DVNFT" for 99 ETH and that the price is fetched from an on-chain Oracle based on 3 trusted reporters. An interesting information as we have found 2 private keys, it might be based for 2 of the 3 trusted Oracle reporters. Maybe we can try to manipulate the Oracles to achieve a specific result in the smart contract.

The Defi Application is consisted of 3 smart contracts:

- Exchange.sol, which is a smart contract doing the exchanges. It utilizes ReentrancyGuard.sol from OpenZeppelin to protect itself from potential attacks related on Reentrancy Attacks. Furthermore it uses events and safe functions such as **safeMint()** to be considered safe for use,
- TrustfulOracle.sol, which is the Oracle which utilizes 3 sources and as we can see, it supports to get data and then sort them by using, it computes the median price (for instance, by having the values 3 1 5, it would first sort these values, which sorts them as [1, 3, 5]. Since the array length is odd (3), it directly returns the middle value, which is 3. This value is the median price of the given array. For an even number of elements, such as [2, 4, 6, 8], the **_computeMedianPrice** function calculates the average of the two middle elements. In this case, the middle elements are 4 and 6, so it computes (4+6)/2=5. This average, 5, is the median price for an even-numbered array.
- TrustfulOracleInitializer.sol, which deploys a new **TrustfulOracle.sol** smart contract with specified data sources, initializes it with given symbols and their corresponding prices, and emits an event with the oracle's address. It leverages constructor parameters to dynamically set up the oracle upon deployment, ensuring initial data is directly integrated.

Therefore, as we control 2 of the Oracles that the smart contract feeds itself with data to categorize and set a price in the NFTs by taking the median, we can change the price of the NFTs to 0:

- **await this.oracle.connect(trustedOracle1).postPrice(tokenSymbol, 0)**
- **await this.oracle.connect(trustedOracle2).postPrice(tokenSymbol, 0)**

Then, we can buy the NFT for 1 Wei as the **Exchange.sol** has a requirement for a price being greater than zero.

Then, as we control the 2 Oracles, we can change the price and increase it to 9990ETH, which is the balance of the deployed **Exchange.sol**:

- **await this.oracle.connect(trustedOracle1).postPrice(tokenSymbol, EXCHANGE_INITIAL_ETH_BALANCE)**
- **await this.oracle.connect(trustedOracle2).postPrice(tokenSymbol, EXCHANGE_INITIAL_ETH_BALANCE)**

Then we have to approve the NFT for exchange and sell it back for a higher price:

- **await this.nftToken.connect(attacker).approve(this.exchange.address, 0)**
- **await this.exchange.connect(attacker).sellOne(0)**

And finally, in order to hide the modification that we have done, we can change the price of the NFTs back to their original price:

- **await this.oracle.connect(trustedOracle1).postPrice(tokenSymbol, INITIAL_NFT_PRICE)**
- **await this.oracle.connect(trustedOracle2).postPrice(tokenSymbol, INITIAL_NFT_PRICE)**

And we have successfully purchased the NFT for just 1 Wei and sold it for 9990ETH. That example was an awesome demonstration to show that:

- Companies and individuals developing smart contracts and decentralized technologies must secure both their centralized and decentralized infrastructure to prevent leaks and potential exploitation, as the potential attacks in the centralized infrastructure could be scaled to the smart contracts deployed in Blockchains,
- Reliance on a single private key for managing decentralized applications, such as Oracles in which are based many smart contracts in order to be functional, could be fatal in case of a leak of the key (on this example was a leak from the web server, but it could pretty easily be a leak related on social engineering attacks such as phishing or gaining access on a computer, browser or wallet). To mitigate such vulnerabilities in critical infrastructure and avoid single points of failure, adopting technologies such as multi-factor authentication is of high importance. The significance of these strategies, especially the role of multi-signature wallets in hardening security, will be further explored in the upcoming discussion on Multisignature Wallets.

6.7. Stable Coins

Due to high volatility of cryptocurrencies, or when there is a need to represent the value of one currency on another blockchain, we can use stable coins.

Stablecoins represent a predefined value from one currency or asset to an another Blockchain and they use various mechanisms to preserve this exchange rate.

For instance, Tether (USDT) is a stable coin that represents the US Dollar (USD) to the Blockchain ecosystem. Tether is pegged 1:1 with the USD. As a result, everyone in Tether's supported Blockchains can turn its volatile crypto assets, especially in high volatility or risky periods, to USDT in order to save its value. The question that possible raises from this is how the 1:1 can be achieved, especially within the blockchain ecosystem.

There are two main approaches to achieve this:

- **Centralized approach**, by creating fiat, crypto or commodity collateralized stablecoins,
- **Decentralized approach**, by developing algorithmic stablecoins.

6.7.1. Fiat-Collateralized Stablecoins

In the case of Fiat-collateralized stablecoins, typically, a company offers a stable token (as we mentioned with USDT), in which this company is required to maintain a reserve of fiat currencies (for example, for every "Cryptodollar" issued, they will reserve an equivalent in real USD).

In reality, these kinds of companies are also holding other assets such as gold, silver even commodities like oil or real assets. These companies in order to promote transparency, are conducting regular audits from independent assurance auditors to prove and verify ownership of these assets.

In the case of Tether, from the report of 31 December 2023 we can find out that Tether has its reserves 84.58% in bash, cash equivalents and other short-term deposits, 0.05% in corporate bonds, 3.62% in precious metals, 2.91% in Bitcoins, 4.95% in secured loans and 3.89% to other investments. Following these and that in terms that they are being independently audited from BDO Italia, promotes transparency in Tether and makes them trustworthy in the crypto community. The Audits that happen in order to prove their respective solvency from reputable independent companies are named Proof of Reserve Audits.

Except Tether, there are a lot of other fiat-collateralizes stablecoins such as the USDC, Gemini Dollar (GUSD), Binance USD (BUSD). Also there are options such as Wrapped Bitcoin (WBTC) which is an ERC20 token pegged 1:1 with Bitcoin.

While many well-known stablecoins are created by reputable companies, there are also many of them that there is a lack of transparency regarding the processes used by some firms to create and issue their stablecoins and peg them with fiat currencies such as USD, euro etc. As these tokens are getting minted in blockchain under a centralized model from a company, it is possible for a company to issue more stablecoins representing fiat crypto or commodity tokens that these that they are really own. As the entire concept relies heavily on trust, it is possible that they could continue issuing more stablecoins until a financial problem occur in the company or until trust is ultimately lost, perhaps due to a legal issue. For instance, what could happen considering if SEC was started investigating for money laundering a company providing centralized stablecoins? In such a scenario, since these stablecoins are based on trust in a specific company, users could lose all of their money if the trust was ended up lost.

In the Ethereum Based world, USDT can be found as an ERC20 token on Ethereum Mainnet, Polygon, Avalanche and Cosmos Blockchain. Except the ERC20 tokens style, USDT is released in other Blockchains such as Solana, Polkadot even Tezos.

Collateralized Stablecoins can also represent real assets such as gold or silver, even other stocks and cryptocurrencies from other Blockchains.

In order to overcome problems like these explained before based on trust in the centralized model of governance, blockchain community has started to develop decentralized models to create stablecoins.

6.8. Decentralized Model: The algorithmic stablecoins

As the name suggests, algorithmic stablecoins are decentralized, fundamentally relying on smart contracts. Algorithmic stablecoins don’t use fiat, crypto or commodities as collateral and instead, smart contracts automatically manage the supply of tokens in circulation by predefined code which has been specially written in order to satisfy specific needs. For instance, in terms of an algorithmic US Dollar, the smart contract will reduce the name of tokens that are available in circulation when the price falls below a predefined threshold.

Conversely, if the token’s price exceeds a certain threshold, then the smart contract issues additional tokens to adjust the stablecoin value with the classic principles of supply and demand.

One of the most known algorithmic stablecoin projects pegged to US Dollars is USD Coin.

While algorithmic stablecoins can provide better transparency as their smart contracts are public, allowing anyone to audit their code and review documentation such as whitepapers, GitHub etc. to identify and analyze how a project works and how they follow their plans. Moreover, as algorithmic stablecoins usually are backed by locked digital assets on a blockchain which are not directly controlled from a centralized company.

Nevertheless, that doesn’t mean that algorithmic stablecoins are without their inherent risks.

6.8.1. The Terra-LUNA crash

The Terra-LUNA collapse serves as a significant case study in the blockchain ecosystem as it was a proof with potential vulnerabilities that could happen in algorithmic stablecoins and end up destructive.

TerraUSD (UST) was an algorithmic stablecoin designed to maintain a peg with the US dollar through the use of algorithms in smart contracts to maintain its value. Unlike fiat-collateralized stablecoins as we explained with USDT, which back their value with USD reserves and other commodities, UST stability mechanism was relied on a balancing act with another token, called LUNA. The ideas was that each UST could be exchanged for 1$ worth of LUNA and vice versa at any time. If the price value of UST fell below 1$, traders could buy UST and swap it for 1$ worth of LUNA. Traders could benefit from this practice, profiting from the difference between LUNA and UST. At the same time, as they swap it for LUNA, they price value of was being increased, keeping the 1:1 peg.

Conversely, if case of UST’s value rose above 1$ then the smart contract automatically minted new UST and then exchanged LUNA for UST, increasing the UST’s supply and as a result adjusting the price back to the normal 1:1 with USD parity.

This innovative approach relied heavily on continued faith in the ecosystem’s stability and growth, assuming that UST demand would remain high and the mechanisms for maintaining the peg was good enough to withstand market pressures.

However, in May 2022, a crisis in Anchor Protocol, a lending platform offering high yields on UST deposit and was, which significantly contributed to UST’s demand, started to lose its peg to US Dollar due to large withdrawals. As the price of UST fell, the designed mechanism from smart contract to sell LUNA to purchase and burn UST intended to reduce its supply and restore the peg, became ineffective without investor's confidence. The market soon was flooded with LUNA causing its price to collapse as a consequence.

This led to a vicious circle where the falling price of UST led to more LUNA being minted at an unsustainable high rate, further devaluing both tokens. After these events, panic selling had continued, reducing further the pegging price. To stabilize the situation, the developers tried to inject liquidity and propose solutions to restore the confidence in the market. However, in the end, the UST lost its peg entirely, losing all of its value.

LUNA's crash not only wiped-out billions in market value but also significantly impacted the confidence in algorithmic stablecoins and enhances the importance of risk assessments in the world of smart contracts and developing incident response plans to minimize those risks except the security of the smart contract. An Incident Response Plan could have played a crucial role in mitigating the impact of crises like this or identify possible scenarios that could had happen. On LUNA's scenario, a transparent incident response plan could have kept stakeholders informed about how they would try to mitigate a scenario like this, promoting more trust in the community and reduce further panic. While an incident Response Plan might not have prevented the collapse it could have significantly mitigated the impact by providing a framework for seeking resolutions.

6.9. Atomic Swaps

An atomic swap allows users to trade cryptocurrencies across different blockchain networks, directly with each other without intermediaries like a traditional exchange. With that process, individuals who own their tokens keep full control over the swap process. Two parties agree to trade their tokens at a mutually agreed-upon amount. In order to achieve this, atomic swaps are being represented as Smart Contracts in both chains. Once they both provide consent, the smart contracts execute the predefined actions. After execution, the transaction becomes irreversible thus if both parties need to swap their tokens back, the have to follow again the process.

As the name suggests, the atomic swaps functions in a way which if the transactions succeeds then it means that the swap has been completed in both chains. As a result, inheriting secure developing standards in atomic swaps are of high importance.

6.10. Bridges

For over 6.000 years, bridges have been the backbone of civilizations, connecting disparate lands, enabling nations and empires to trade with other empires, fostering economy and unity. Over the years, bridges formally and informally have been used for a lot of operations, including in global relations, "building" cultural bridges between diverse cultures and nations. With the expansion of the word of bridges, bridges have finally arrived in the Blockchain ecosystem in order to provide interoperability by connecting different Blockchains together. As previously detailed, Blockchain Networks are designed to operate independently thus they cannot send data or communicate with other Blockchain Networks. Oracles can be utilized to communicate with outside world, however having a process to send assets from one Blockchain to an another is of high importance, providing more scalable solutions in the Blockchain ecosystem. Bitcoin and Ethereum are the most well known Blockchains, sharing over 70% of entire's Blockchain market share. The problem that has discussed further on this thesis especially for Ethereum are the gas fees. As Ethereum is the most widely used

Blockchain for utilizing smart contracts, the gas fees especially in bull runs can be further increased rapidly. Even in distance of some minutes, the gas fees can be very volatile as they work like in the market of need and demand.

For this reason, in order to avoid network congestion and thus higher transaction fees, it would be beneficial if we have the opportunity to represent some assets to other Blockchains which are more scalable or with less over-demand, optimizing the network traffic. For this reason, we could make transactions and move funds and assets more easily in less cost.

This is precisely where Blockchain Bridges came for.

Blockchain Bridges are divided into two main categories:

- **Trust Based Bridges**, which as we have discussed to other chapters for oracles and stable coins, they are based to a centralized company controlling a bridge sending data and assets from one Blockchain to another. The centralized service acts as the custodian, ensuring the bridge's functionality and security. Due to their nature, Trust-Based Bridges offer rapid transfers of assets, but they have to use intermediates in order to operate, having the risks associated with the common single point of failure.
- **Trust-less based Bridges**, which are decentralized, provide connection between blockchain networks, allowing the transformation of the assets without intermediaries. Trust-less bridges function independently of any central intermediaries, and they instead they are utilizing smart contracts for the authentication of transactions and the assurance of security by using algorithms on smart contracts to move assets from one blockchain to an another. However, due to their decentralized nature, trust-less based Bridges tend to be slower compared to Trust-Based Bridges.

Prominent examples of Trust-less Blockchain Bridges communicating with Ethereum are:

- **Binance Bridge**, connecting Ethereum Network with Binance Smart Chain,
- **Avalanche Bridge**, connecting Ethereum Network with Avalance,
- **Polygon Portal Bridge**, linking Ethereum Network to Polygon POS and Polygon zkEVM,
- **Wormhole and Sollet**, connecting Ethereum Network with Solana.

These bridges are only a few examples of Trust-less Bridges. Numerous bridges are additionally available across a wide range of blockchain networks.

Blockchain trust-less Bridges in practice in the Ethereum based world, are working with smart contracts. Considering a scenario where a Blockchain A with A token and a Blockchain B with B token, a smart contract representing B token as an ERC-20 token exists on Blockchain A, and reversely a A token is represented as an ERC20 on Blockchain B.

When a user intends to transfer A tokens from Blockchain A to Blockchain B,

- The smart contract on Blockchain A locks a specific amount of A tokens in the and sends a message through a bridge to Blockchain B to inform it with the amount of A tokens that have been sent,
- Subsequently, Blockchain B generates an equivalent amount of A tokens as ERC-20 tokens to Blockchain B so that the user could be able to trade A tokens to Blockchain B.

If the users want to send these A tokens back to Blockchain A,

- The user uses a smart contract on Blockchain B which either burn the ERC-20 representation of A tokens or lock them. This action ensures that the tokens are removed from circulation on Blockchain B,
- The bridge then communicates with Blockchain A to unlock the precise amount of A tokens.

While this process can be extremely beneficial for trading assets that are typically incompatible across different Blockchain networks, or even trading an asset externally from its main Blockchain to an another Blockchain or L2 scaling solution, often resulting in lower transaction fees, similar to physical bridges which face threats from natural disasters or human adversaries, Blockchain Bridges are also susceptible to numerous challenges and potential vulnerabilities that could happen which can damage both Blockchain Networks and creating new critical issues and then transits these issues from one Blockchain to an another. As Blockchain Networks become increasingly interconnected through bridges, the risks associated with cross-chain attacks would be escalated significantly. Periodic Auditing of code functionality would be crucial for safeguarding decentralized applications, especially Defi applications or Decentralized Exchanges which are using bridges to enable cross-chain trading and liquidity.

6.10.1. Broken Access Control on a Bridge

We have a smart contract called SimpleBridge.sol which represents a simple Trust-less Bridge in Ethereum as a Solidity written smart contract which allows for the deposit and withdrawal of funds, but it includes a critical function intended for administrative use which lacks proper access control measures. An attacker could exploit this to withdraw funds without proper authorization. After auditing the smart contract's source code, we notice that the function **updateBalance()** is intended to adjust user balances in specific scenarios, such as correcting errors or updating balances after certain off-chain or cross-chain operations. However, without proper restrictions on who would be able to call this function, an unauthorized user could maliciously set their balance to a higher value and withdraw funds that he didn't deposit, draining the smart contract's assets and destroying the bridge between the 2 blockchains.

A potential solution on this could be to set an access control check such as a modifier which would enforce that only the owner of the smart contract might be able to apply modifications of the smart contract. Nevertheless, it is a significant challenge in constructing Trust-less Bridges is the reliance on an administrator to have the proper access to modify or update the smart contract's balance. In order to limit the power from a user to a specific smart contract, even a set of decentralized applications or Defi platforms and promote transparency in governance, we can use multisignature wallets.

6.11. Multi Signature wallets

In the real world with the typical company's IT infrastructure creating an Information Security Management System (ISMS) involves enforcing strict Access Control policies such as two-factor authentication (2FA) or using restricted accounts. These measures are implemented to enable administrators to perform their tasks while preventing malicious or unintended events that could disrupt business continuity to happen.

In Blockchain world in order to promote better transparency, we can use multisignature wallets. A multisignature (multisig) wallet is a type of digital wallet which requires multiple signatures or approvals from a predefined group of owners to execute transactions. This also applies to smart contracts, where executing a function or changing functionality requires, multiple approvals. Multisignature increase security by distributing trust among several parties, reducing the risk of unilateral actions due to compromised keys or malicious intent from a specific user.

By applying multisignature behavior to smart contracts, owners can propose changes in the balance of the smart contract. This proposal must then be approved by other predefined owners of the smart contract. Only if all of them approve the changes, will the smart contract accept the proposal. The multisignature approach adds a layer of governance, making it suitable for bridge operations that need to ensure high levels of security and consensus.

In real blockchain scenarios, applying multisignature mechanisms to smart contracts can be extremely beneficial in providing transparency and trust in various projects, especially in Defi as users would know that on this project a stubborn or a malicious user could not maliciously modify the blockchain data. Furthermore, it reduces the risk of funds being stolen or misappropriated by requiring consensus among multiple parties for critical actions, thus promoting more transparency and decentralization. As we understand from the previous explanations, avoiding single points of failure is of high importance for the broader acceptance of technologies running in Blockchain Networks, as it significantly impacts market confidence, making the investors having second thoughts for even trustworthy projects.

One of the most famous implementations of Multisig wallets is MultiSigWallet.sol whose code can be integrated into other smart contracts. Additionally Safe.global provides wallets from Ethereum Based networks such as AAVE, Polygon, Ethereum mainet , Arbitrum etc. However, as smart contracts or application wallets', mutlisignature wallets must be audited for security as there are many potential vulnerabilities that they might be included.

It should be mentioned that OpenZeppelin offered its own version of multisignature smart contract called gnosis-multisig however it has been archived on GitHub and it is no longer supported. Therefore, it is not recommended for use, as the code may contain vulnerabilities.

6.11.1. The Parity Wallet hack (exploiting and bypassing multi signature verification)

While multi signature wallets can be very beneficial for the transparency and the access control configurations in a smart contract to reduce the risks associated with an executive to abuse his rights or to be stolen, a bad implementation in the way that multisignature wallets work could end up fatal as the attack could potentially bypass the multisignature authentication and then enforce its modification on the smart contract. The Parity Wallet hack was one kind of these attacks back in 2017 and the exploitation of the vulnerability ended up with stealing more than 150,000 ETH. The vulnerable smart contract was called Parity Multisig Wallet in the version 1.5+.

To describe the vulnerability shortly, the MultiSig wallet attack exploited a vulnerability which was based from the misuse of the **delegatecall()** function (we have seen in the previous sections a simplified example of how delegatecall attacks can be executed), combined with improperly secured initialization logic. In order to achieve this, attackers sent two transactions to the affected smart contracts.

The first transaction called a function called **initWallet(),** allowing the attackers to misappropriate the ownership by reinitializing the wallet's ownership to an address under their control. The **initWallet()** function, intended to abstract the constructor logic into a library for reusability and gas efficiency, it lacked safeguards mechanisms against subsequent calls.

The vulnerability was triggered through a delegate call to an external library, which treated all functions as public due to the fallback function in the wallet smart contract, forwarding in practice the calls via the **delegatecall().** That means that the vulnerability made anyone able to call all the public functions from the library. This oversight enabled the attacker to call **initWallet** directly, changing the smart contract's **m_owners** to their address and setting the required confirmations to one. With that method, in practice the multisignature logic implementation of the smart contract have been bypassed and was able to accept changes only by one user. The multi factor authentication theme introduced in Smart Contracts to reduce the access control attacks in broken. In the next steps, the subsequent transaction utilized the execute function to transfer all funds to the attacker-controlled account, exploiting the newly acquired sole ownership and thus the attacker successfully have stolen 153.037 ETH.

This incident is a proof that the common vulnerabilities that we have described in the start of this master thesis should not be avoided. Security Awareness training in the developers and regular security audits in of high importance in order to secure smart contracts and avoid common attacks. Multisignature wallets are a very useful concept to secure your smart contracts however as we have seen, a critical design flaw was enough to exploit the smart contract and steal the funds from a single attacker. For this reason, designing smart contracts in terms of policies and procedures with a detailed documentation and a logic and data flow can be helpful to identify potential vulnerabilities and understand the logic easier to avoid bad coding practices.

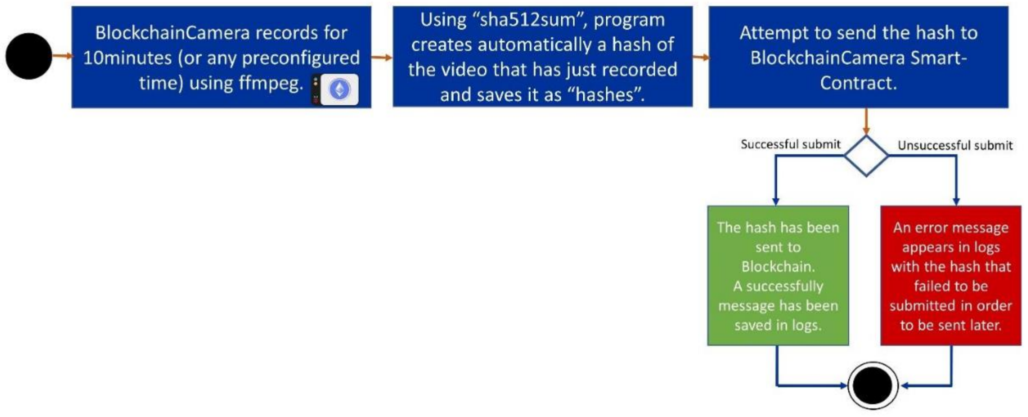

BlockchainCamera records for 10minutes (or any preconfigured time) using ffmpeg.
Using "sha512sum", program creates automatically a hash of the video that has just recorded and saves it as "hashes".
Attempt to send the hash to BlockchainCamera Smart-Contract.
Successful submit
Unsuccessful submit
The hash has been sent to Blockchain. A successfully message has been saved in logs.
An error message appears in logs with the hash that failed to be submitted in order to be sent later.

Figure 10 An activity chart showing how an application sends hashes to Blockchain. An activity chart can be beneficial for showing the designed logic behind an application.

The attack might have been avoided by either keeping the constructor logic within the main smart contract instead of separating it into a library contract or, more effectively, by not using the **delegatecall()** for purposes like this.

6.12. Upgradable Smart Contracts

While smart contracts have been designed as a technology to execute predefined tasks considering the examples discussed before, we see that if they are developed with variables or functions that interact with other external smart contracts, in practice, it is possible to change or manipulate the functionality of a smart contract, as we have discussed with **delegatecall()** etc.. Upgradable smart contracts introduce adaptability to the traditionally immutable nature of smart contracts, gaining attention from companies such as OpenSea as the future of the smart contract technology. Nonetheless, the flexibility of upgradability raises further security and governance concerns.

In the standard model, smart contracts are immutable once deployed, ensuring trust in decentralized applications, including DeFi applications as it is possible even the creators to be prevented from altering the value of a smart contract. On the other hand, despite security audits and checks, there is a high possibility that a security flaw could be overlooked and make into production. Which means that in this case, a vulnerable DeFi application could be addressed and no one would be able to stop the functionality of the vulnerable smart contract, if it wasn't developed in that way (for instance, with the example of **self-destruct()**, introducing risks associated with trust.

It becomes apparent that while immutable smart contracts are valuable, in some projects, especially larger ones, the capability of upgradability can be beneficial, in combination with multiple authentication schemes, implementing DAOs and a transparent policy and procedure for upgradability.

Upgradable Smart Contracts offer a solution, allowing modifications to be made in smart contracts without needing to transfer activities to a new address.

There are several Ethereum improvement proposals associated with upgradable Smart contracts such as:

- EIP-1538: Transparent contract standard
- EIP-1822: Universal upgradeable proxy standard (UUPS)
- EIP-1967: Proxy storage slots
- EIP-2535: Diamonds, multi-facet proxy.

These smart contracts are designed in a way that separates data into another smart contract from the smart contract having a specific logic. It could be described as in the container's technology such as Docker and Podman where we have an image having all useful files to be functional, a volume file storing all user's data and settings and the combination of them is creating a functional container.

In the logic smart contract, functions execute predefined actions, while in the data smart contracts, data are stored in variables or data structures. While this strategy supports upgradable smart contracts, as we maybe understand that the constant calls between logic smart contracts and storage smart contracts could require higher gas fees than if contained within the same smart contract.

A more modern approach of upgradable smart contracts in a way that does not consume excessive gas fees, is based on proxy smart contracts. Users interact with a proxy smart contract which forwards their calls to the logic smart contracts. Due to the design of the proxy-based upgradability, the logic smart contracts do not store any user's data. All that is required is to replace the old address of the logic smart contract in the proxy smart contract with a new one, as it was previously described for containers being used in the real world. The Proxy Contract would use **delegatecall()** to call the logic smart contract.

With this design, developers are able to fix undetected vulnerabilities in the smart contracts that made it into production, without interrupting the use of the smart contract. Furthermore, it saves gas fees as it is more efficient to replace an address than replacing the entire smart contract and having to move user's data during the upgrade, eliminating migration issues. However, as we can understand, this model relies on trust and transparency. Developers of upgradable smart contracts must provide comprehensive documentation and show clear data and logic flaw charts considering how the smart contracts work, including how and where changes are applicable. For instance, an upgradable smart contract which supports modifying tokens in a way that can add a time lock, as this was described in the previous smart contract examples, could be considered as high risk. Except the potential malicious acts from a developer, as the example that was well explained with **delegatecall()**, a missing call could initialize huge security risks in the smart contract if the upgrades goes to production without being noticed.

Moreover, modifying the storage layout during an upgrade could result in storage collisions between implementation versions in the event of a major change in the logical implementation leading to unexpected or unintended results.

Finally, proxy smart contracts could reduce fragmentation and be able to track the changes in the philosophy of a smart contract. However, a single malicious change in the proxy smart contract, could be fatal for user's data and thus user's assets/

Many of the proposals mentioned previously have been discontinued as better ideas have been implemented, requiring less gas fees and better upgradability standards.

For instance, EIP-2535 is a proposal which is still being promoted as it uses smart contracts called diamonds, which allow external functions used other smart contracts called facets. Facets are independent smart contracts working separately from the diamond smart contract and can share between them functions, libraries and state variables. Moreover, EIP-2535 also enhances transparency as it enables tracking the history of changes, promoting more trustworthiness in the project in a way which unusual behavior can be notified and addressed and they support multi-signature smart contracts by default, further enhancing transparency. Additionally, diamond smart contracts can be set as immutable or upgradable, allowing for tracking all changes or modifying specific functionalities.

We can understand that upgradable smart contracts can be extremely beneficial for developing and scaling decentralized applications. Nonetheless, conducting smart contract auditing after each change is essential to build trust within the community. Access Control and transparency in the design and the flow of the data could also reducing the risks as the governance model and the logic behind the smart contract could be much easier being understood and avoid critical issues. Furthermore, especially in larger projects, event logging could also contribute to preventing security incidents, or in case they occur, to finding ways to respond and mitigate the risks.

Chapter 7: Event Logging in Smart Contracts

In the rapidly evolving world of Blockchain Technology and Smart Contracts, although smart contracts are protected by their Blockchain immutability once deployed and cannot be directly modified, the examples that we have been discussed on this master thesis we have seen multiple possibilities, from broken access control mechanisms to the execution of functions not as intended during development or even by using DDoS Attacks on smart contracts. While the Blockchain enforces the consensus rules, the development of smart contract hides numerous potential vulnerabilities which can be exploited by malicious actors and steal the assets from a smart contract, make it unavailable to communicate, or even change its ownership. In this chapter we will discuss about event logging in smart contracts and how they can be used to further secure smart contracts to identify potential attacks and abnormal behavior.

These challenges are not unfamiliar to the information security world. In the conventional businesses, Event Logging serves as a crucial methodology enabling Information Security consultants and System Administrators to identify issues, potential threads or malicious acts from events, processes, or operations within an organization's system. Software and systems record all activities related on security or potential bugs that can lead to crashes and then, notify the employees (for instance, system administrators or Chief Information Security Officer) of the company to take significant measures. Companies write well detail Event Logs (or Audit Log) Policies which explain how logs should be maintained and audited, tested and also offer a well detailed Incident Response Plan which sets the roles of the incident Response Plan and explains how each member of the Plan should behave in case of an incident. Not all incidents share the same criticality, therefore, a varied approach might be necessary, depending on the incident's severity in systems (for instance a malware or ransomware attack in the company's network), infrastructure (Physical Attack measures), what they attacker might be access or alter, even if it happens on chain.

For instance, within enterprise IT environments, Security Information and Event Management (SIEM) systems are used to assess and rank the severity of event logs. SIEM systems play a critical role in modern cybersecurity defenses by aggregating, analyzing, and correlating data from various sources within an organization's IT infrastructure. If they have been correctly configured, they notify the predefined people automatically in case of an incident to trigger an as fast as possible act to prevent an incident to peak or expand to other systems of a company. Furthermore, companies usually establish Security Operations Center (SOC) teams or engage third-party services that offer 24/7 inspection of company's infrastructure and they are responsible for the rapid assessment and mitigation of threats to minimize damage and prevent future attacks, ensuring the security and integrity of information systems.

Having explain this, it becomes clear that event logging mechanisms for identifying incidents in the Blockchain World, will become increasingly common as the market expands.

In the Ethereum world, event logging is achieved through **events** in Solidity programming language, which emit logs from the smart contracts. These logs are stored on the Ethereum Blockchain, providing a transparent and immutable record of actions that can be observed and analyzed by external entities.

Event logging in smart contracts is important for promoting transparency and auditability as they can be used to for recording significant actions within a smart contract and identify potential issues and risks earlier. For instance, logs can be emitted for access control changes or significant alterations to a smart contract's logic variables (for instance, by minting new tokens on an ERC-20 token smart contract). Furthermore, we should consider that typically storing data on Blockchain is very expensive, however in terms of event logs, they have been designed to be gas-efficient compared to storing other data directly on-chain as events are not accessible from within the smart contract once emitted. That means while a smart contract can emit events from a smart contract to log data or actions, the smart contract itself cannot access or read these logs after they're emitted. Utilizing these event logs, developers or security professionals can establish real-time alerting systems, enabling them to notify them when a significant act is occurring, enabling rapid responses to potential issues or suspicious activities.

Although smart contracts are often described as entire "decentralized", many projects allow developers to modify certain variables in the smart contracts as we have seen to previous examples, meaning that for most projects, transparency, trust and the belief is what it shapes the projects, at least for their first steps, what they eventually become. Event Logging omits in Ethereum can be integrated with external systems or platforms to make the accessibility and readability of logs and alerting systems for all users by analyzing common patterns. Moreover, they can be programmed to trigger automated responses or alerts. For example. Alchemy Notify is a webhook based service that sends notifications to users when an event occurs on-chain in well-known Ethereum Based Blockchains.

In Incident Response, a more centralized approach might be necessary to secure smart contracts effectively. As we have discussed from the conventional industries, Incident Response Plans typically predefine mechanisms and actors which have to address and mitigate the impact from security incidents. Due to the immutable nature of Blockchain, the strategies that have to been follows for incident response in smart contracts can be slightly different.

While Response Plans should be developed alongside the white paper or any documentation during the design phase of the smart contract, to maintain effective Incident Response Plans, companies have to conduct annual tests or tests with each major change to evaluate their effectiveness. To identify potential attacks, there are many tools available inspecting Blockchain Networks and Smart Contracts to identify common pattern and potential threats. In the next chapter, we will briefly discuss some of these tools that are available. As it happens with most of the tools that have been analyzed, shown, or mentioned on this master thesis, the ideas that are being implementing on Blockchain Networks is so rapid right now that is practically impossible to keep up with all of them, but many can be referenced for future considerations.

7.1. Forta Network

The Forta Network a range of tools designed to support in Event log Analysis and Incident Response.

Forta Threat Intelligence includes many tools such as Scam and Spam Detector or Sniffer Rugg Pull Detector, Sybil Defender, Attack Detector among others. These tools analyze data through bots in order to identify common patterns. For example, they can analyze smart contracts code or even token's code to detect potential malicious activities and in the event of such activities, notify the asset owners.

Forta Threat Intelligence is accessible across numerous Ethereum Based Blockchain Networks including Ethereum Mainnet, Polygon, Optimism, Avalanche and more.

In this master thesis we will discuss further about Attack Detector Feed, as it specifically focuses on analyzing events from smart contracts.

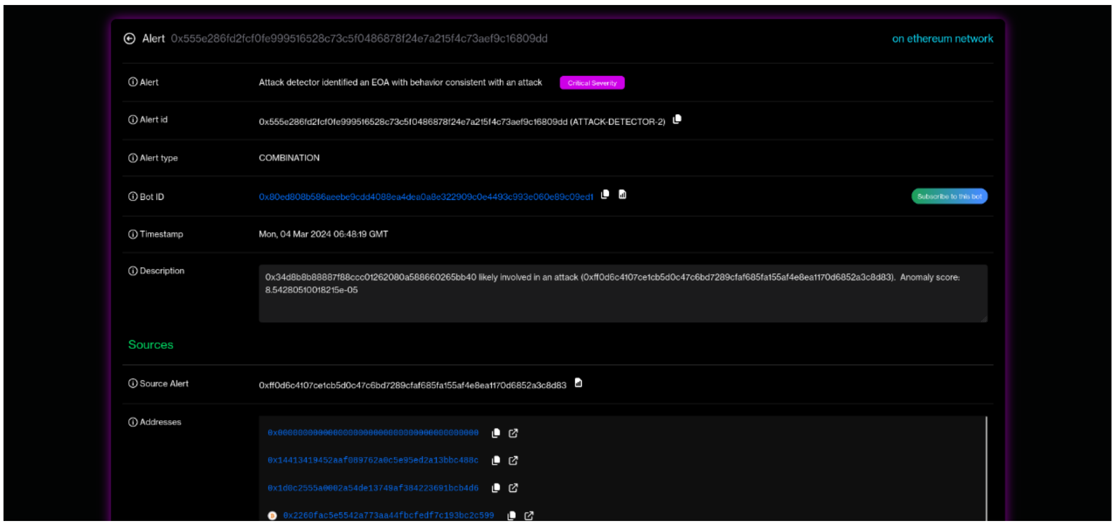
7.2. Forta Attack Detector Feed

The Attack Detector Feed functions as a special tool which can be described as a SIEM but for Ethereum Based Blockchain Networks. Attack Detector Feed can provide real time alerts from smart contracts, prior to, during or directly after the attack, as the official site suggests. The main features of Attack Detector Feed explained in the official web site are these:

- **Real-Time Protocol Attack Detection:** The tool identifies potential attacks by analyzing past alerts associated to a common address, addresses associated from previous attacks and thereby enhancing the precision of alerts. Additionally, by aggregating past low-precision alerts from base bots and mapping them to the attack stages, the tool effectively filters out false positives, leaving more space for real true positive attacks to be identified and addressed,
- **Four Attack Stages:** Alerts are categorized into four stages of an attack: Funding, Preparation, Exploitation, and Money Laundering/Post Exploitation, enabling and clustering into a more comprehensive monitoring and detection,
- **Anomaly Detection Approach:** Alerts are scored for anomalies, with scores mapped to specific attack stages. Critical alerts are emitted based on a combination of minimum anomaly scores across all stages, the total number of alerts, and specific thresholds for anomaly scores.

In order to achieve better outcomes, Attack Detector Feed utilizes 2 Heuristic Approaches in order to enhance the detection:

- Utilizing highly precise alerts from specific base bots, triggering a critical alert when combined with another alert from a different stage or another precise alert,
- Emitting a critical alert when an alert is raised in each of the four attack stages for a common cluster of addresses, ensuring a comprehensive coverage of potential attack vectors.


Alert 0x555e286fd2fcf0fe999516528c73c5f0486878f24e7a215f4c73aef9c16809dd

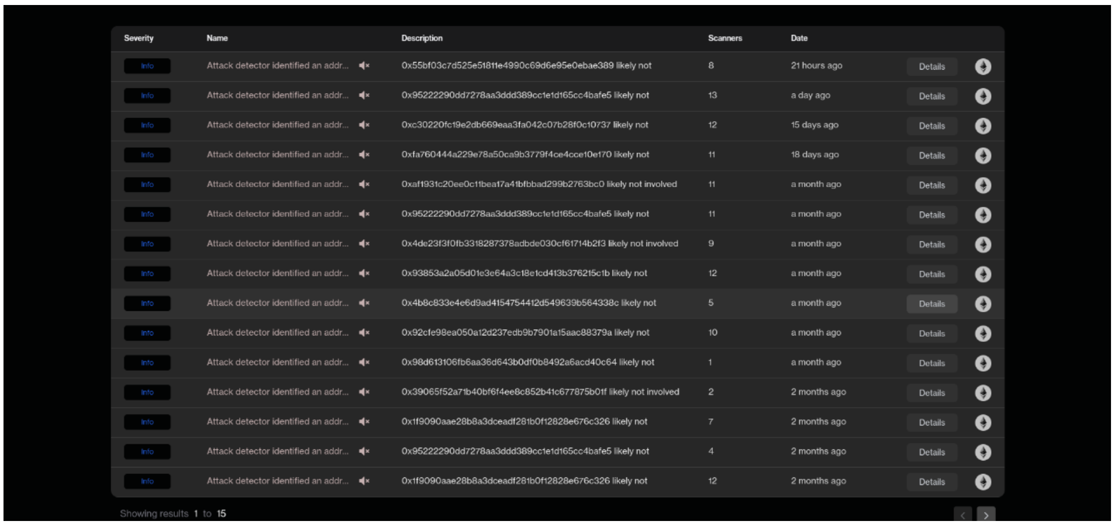
on ethereum network

Alert	Attack detector identified an EOA with behavior consistent with an attack Critical Severity
Alert id	0x555e286fd2fcf0fe999516528c73c5f0486878f24e7a215f4c73aef9c16809dd (ATTACK-DETECTOR-2)
Alert type	COMBINATION
Bot ID	0x80ed808b586aeebe9cdd4088ea4dea0a8e322909c0e4493c993e060e89c09ed1
Timestamp	Mon, 04 Mar 2024 06:48:19 GMT
Description	0x34d8b8b88887f88ccc01262080a588660265bb40 likely involved in an attack (0xff0d6c4107ce1cb5d0c47c6bd7289cfaf685fa155af4e8ea1170d6852a3c8d83). Anomaly score: 8.54280510018215e-05

Sources

Source Alert	0xff0d6c4107ce1cb5d0c47c6bd7289cfaf685fa155af4e8ea1170d6852a3c8d83
Addresses	0x0000000000000000000000000000000000000000 0x14413419452aaf089762a0c5e95ed2a13bbc488c 0x1d0c2555a0002a54de13749af384223691bcb4d6 0x2260fac5e5542a773aa44fbcfedf7c193bc2c599

Figure 11: A critical alert shown from Attack Detector Feed.



Severity	Name	Description	Scanners	Date		
Info	Attack detector identified an addr...	0x55bf03c7d525e51811e4990c69d6e95e0ebae389 likely not	8	21 hours ago	Details	
Info	Attack detector identified an addr...	0x95222290dd7278aa3ddd389cc1e1d165cc4bafe5 likely not	13	a day ago	Details	
Info	Attack detector identified an addr...	0xc30220fc19e2db669eaa3fa042c07b28f0c10737 likely not	12	15 days ago	Details	
Info	Attack detector identified an addr...	0xfa760444a229e78a50ca9b3779f4ce4cce10e170 likely not	11	18 days ago	Details	
Info	Attack detector identified an addr...	0xaf1931c20ee0c11bea17a41bfbbad299b2763bc0 likely not involved	11	a month ago	Details	
Info	Attack detector identified an addr...	0x95222290dd7278aa3ddd389cc1e1d165cc4bafe5 likely not	11	a month ago	Details	
Info	Attack detector identified an addr...	0x4de23f3f0fb3318287378adbde030cf61714b2f3 likely not involved	9	a month ago	Details	
Info	Attack detector identified an addr...	0x93853a2a05d01e3e64a3c18e1cd413b376215c1b likely not	12	a month ago	Details	
Info	Attack detector identified an addr...	0x4b8c833e4e6d9ad4154754412d549639b564338c likely not	5	a month ago	Details	
Info	Attack detector identified an addr...	0x92cfe98ea050a12d237edb9b7901a15aac88379a likely not	10	a month ago	Details	
Info	Attack detector identified an addr...	0x98d613106fb6aa36d643b0df0b8492a6acd40c64 likely not	1	a month ago	Details	
Info	Attack detector identified an addr...	0x39065f52a71b40bf6f4ee8c852b41c677875b01f likely not involved	2	2 months ago	Details	
Info	Attack detector identified an addr...	0x1f9090aae28b8a3dceadf281b0f12828e676c326 likely not	7	2 months ago	Details	
Info	Attack detector identified an addr...	0x95222290dd7278aa3ddd389cc1e1d165cc4bafe5 likely not	4	2 months ago	Details	
Info	Attack detector identified an addr...	0x1f9090aae28b8a3dceadf281b0f12828e676c326 likely not	12	2 months ago	Details	

Showing results 1 to 15

Figure 12: Alerts from Attack Detector Feed from Ethereum Blockchain

Chapter 8: Risk Management and Incident Response in Smart Contracts and Development and in Production

Smart contract security is a noble effort at multiple levels. In addition to monitoring event logs from the decentralized applications running in Blockchain Networks, companies will need to evaluate the risks associated with their smart contracts. By managing these risks, companies can gain a better understanding of the potential risks and design plans on how they should react in the event of an incident. These plans are known as Incident Response Plans.

Incident Response Plans must be based on Risk Assessments conducted by the company or developers supporting each decentralized application.

Risks associated with smart contract security as we have thoroughly described in this master thesis are everywhere as these protocols evolve the immutability and transparency of processes in multiple DeFi applications present a point of contention in several projects. Following inferior quality of coding practices remains a significant issue in the Blockchain world as many projects are poorly written and they are not following business best practices.

In order to mitigate security vulnerabilities in smart contracts and how potential risks have to be communicated and addressed in an organization, we are conducting Risk Management strategies to evaluate the security and potential risks associated with this smart contract or Defi Application.

Risk analysis in the smart contract world would be required to record several things which may depend on several characteristics, including but not being limited to:

- How Decentralized a smart contract or a decentralized application is intended to be? On this question, a detailed documentation such as a GitHub README.md file, reading the docs or a white paper could be ideal to identify and design how the governance would operate on this smart contract. This can be also beneficial for promoting more transparency in a project. In order to answer this question, we should also considerate how access control would work on the smart contract and use this to identify the potential risks,

- How are smart contract vulnerabilities identified and addressed during the development lifecycle? On this question, we are trying to examine and evaluate the process of periodically scanning for security vulnerabilities in the code of smart contracts for vulnerability scanning, code reviews, and audits to detect and rectify security flaws before deploying the smart contract on a Blockchain.
- What risk mitigation strategies are adopted to handle known smart contract vulnerabilities such as reentrancy, integer overflow, and oracle manipulation? The focus here is on specific countermeasures that can be implemented in smart contracts (as we have shown before, by utilizing OpenZeppelin's ReentrancyGuard for Reentrancy Attacks),
- How does the project manage and secure cryptographic keys and sensitive data associated with the smart contracts? This question examines the procedures and technologies used to secure cryptographic keys and sensitive information from unauthorized access and leaks, not only in the smart contracts, but also in the entire company (securing company's infrastructure and information systems, implementing event logging and alerting systems, hardening access controls etc.),
- In the event of a security breach or the failure of a smart contract, what incident response mechanisms are being followed? This question is related on understanding the preparedness and response strategies for potential security incidents, including mechanisms for bug bounties, smart contract upgrades, or even emergency stops,
- How is the upgradeability of the smart contract is managed, and what risks does it introduce? This question seeks to explore the governance and technical mechanisms for contract upgrades and the associated security implications of such changes.
- How does the smart contract interact with external data sources or other smart contracts, and what risks do these interactions pose? The focus here is on the security and reliability of external dependencies, including oracles and other smart contracts or libraries, interfaces etc., which could introduce vulnerabilities as we have seen in the previous examples, especially with flashloans and ERC20 tokens,
- What measures are in place to ensure user privacy and data protection in compliance with regulations such as GDPR in European Union? This explores the approaches taken to protect user data and ensure regulatory compliance within the smart contract and associated decentralized application.

These fundamental questions are necessary in evaluating the risks associated with smart contracts and design effective Incident Response Plans in order to find measures to reduce the residual risks.

The way in which companies and developers address these questions or uncover related questions based on smart contracts' operation and interaction with other smart contracts is crucial for designing secure smart contracts.

After evaluating the risk and prioritizing the most critical findings, companies must find ways to address these risks or if it is not possible, to minimize the risks associated. This process is called Risk Treatment. In the typical companies, it is standard practice for companies to design and implement policies and procedures based on Risk Management and Risk Treatment in order to follow a predefined plan on how the Risk must be treated. Moreover, it is a standard practice for companies to create inventory lists with all systems and users associated with company's operation and especially how developers work to be prepared in case of critical incidents. They should also associate specific information systems with specific people in order to know their duties as well as to record their job responsibilities in combination with segmentation of duty. With these measures, in case of an incident:

- Companies have recorded their potential risks and have categorized the issues related on their criticality to ensure a prioritized and effective response,
- In case of an incident, each individual is aware of their responsibilities, as roles are clearly defined, including their specific system oversight areas (for instance, preventing an attack on the smart contract development environment aimed at deploying a malicious upgrade),
- Establishing immediate containment protocols, such as suspending the affected smart contracts or temporarily limiting the use and access to minimize further damage while the incident is investigated. This approach is crucial for incidents like this as a malicious alteration could compromise the smart contract in the phase of deployment in a Blockchain,
- Implement detailed analysis and forensic procedures for conducting a thorough analysis of the incident, including blockchain forensics and smart contract audit trails, to understand the source, method, and extent of the breach. This process should identify the specific vulnerabilities exploited and any data or assets affected, providing essential insights for preventing future incidents,
- Following incident resolution and the restoration of standard operations of the smart contract or decentralized application, conduct a comprehensive review to assess the effectiveness of the incident response. Update risk assessments and security protocols based on lessons learned and strengthen smart contract and development environment security measures to guard against similar attacks in the future. Enhancements may involve updating code review procedures, the security of the development environment and company's infrastructure, enhancing security training for developers, and integrating more security tools and practices into the development lifecycle.

8.1. Fund Recovery

The Incident Response plans could include a lot of elements, including infrastructure and company transparency, how they should operate in case of an incident inside the company or even in the Blockchain Network that a decentralized applications are deployed. A critical part in designing Incident Response Plans is to design ways and procedures in order in case of a critical incident which the attacker could withdraw smart contract's funds, to find a way to stop it, or simply limit the withdrawal. By utilizing tools those from the Forta Network for smart contract activity inspection or employing a Security Operations Center (SOC) for 24/7 monitoring of blockchain and smart contract state activities, can support early attack detection. However, identifying an attack does not guarantee the ability to restrict these actions immediately. On the other hand, if somehow a malicious attacker could achieve to gain access on the smart contract, the funds are in danger.

Therefore, during the development phase of a smart contract, various measures can be integrated into its functionality to mitigate the actions a malicious attacker can execute until the threat is identified or rectified (e.g., through upgradable smart contracts). In the next bullets will be represent some ways to protect funds in case of an exploitation:

- Access Controls such as multiple signature wallets have been described as ways to restrict the changes in the state of a smart contract and reduce the single point of failures,
- Implementing freezing mechanisms which in case that a vulnerability is being addressed, it could pause the smart's contract functionality until a patch version could be deployed, detected vulnerabilities or attacks, preventing further transactions until issues are resolved. This mechanism can be crucial for stopping unauthorized fund transfers as soon as a breach is detected,
- Time Locking mechanisms, described previously in an attacking scenario, which serves as a last resort to transfer the remaining balance to a specified address and remove the smart contract from the blockchain,
- The Self-Destruct mechanism, while controversial due to abuse potential serves as a last resort to transfer the remaining balance to a specified address and remove the smart contract from the blockchain. This mechanism should be used cautiously, with explicit safeguards to prevent malicious use,
- Community Consensus (in open-source public Blockchains): If the smart contract is open source and governed by a community or a DAO (Decentralized Autonomous Organization), the community might collectively decide on a course of action, potentially including a recovery process. This, however, requires extensive agreement and coordination among the participants, as well as clearly written documentation of how governance is being achieved.

A few years ago, there was even a controversial Ethereum Improvement Proposal (EIP-867) called Standardized Ethereum Recovery Proposals that was submitted, which in practice could be described as a way of community consensus.

8.1.1. EIP-867 - Standardized Ethereum Recovery Proposals

EIP-867 aims to provide a standardized method for recovering certain classes of lost funds on the Ethereum blockchain. This mechanism particularly focused on scenarios where there is consensus among affected parties regarding the rightful ownership and recovery of assets, thus facilitating prompt and secure resolution. The methodology relies on establishing clear approval standards, designing a common format for detailing corrective actions, and setting guidelines for client implementation to apply actions at a designated block. The idea was to propose a way to recover lost funds and at the same time to ensure transparency, consistency, and minimal risk in fund recovery processes, with resolutions for non-controversial loss incidents within the Ethereum ecosystem.

However in practice, many opposed such a proposal and argued that the reason why Blockchain Networks have been created and utilized decentralized applications was to enforce immutability and consensus mechanisms to enforce immutability and consensus mechanisms, ensuring that the data and smart contracts remain unaltered once deployed. While implementing a proposal such as EIP-867 might support people from losing funds it could infringe on the core principles of blockchain technology. The proposal generated significant debate within the Ethereum community, and the lack of consensus played a critical role in halting the proposal's progress.

8.2. Insurance in Smart Contracts

In conventional companies, especially the larger ones, designing security measures and conducting risk assessments to analyze potential risks are common practices. It is a widespread approach to utilize insurance as a methodology to further mitigating risks and enchasing trust among clients. For instance, many companies are enforcing a Supply Chain or Third Paty Management Policy which obliges the associated partners to have insurance, among other strict cybersecurity measures. Implementing insurance in Defi applications or other decentralized applications can be beneficial for blockchain companies as except their transparency to build trust, they add another layer of trustworthiness as in case of their security model fail, the insurance company could return the value of the lost assets associated with this network. Insurance emerges as a key instrument tool for mitigating risks and strengthening trust within the ecosystem as it offers financial protection against vulnerabilities, operational disruptions, and regulatory uncertainties, insurance not only safeguards projects' financial health but also bolsters confidence among investors, users and developers.

As with many concepts discussed in this master's thesis, deviations arise in the form of:

- **Centralized insurance**, as it is recognized in the contemporary world, is offered by reputable regulated companies. Centralized entities are often well-equipped to navigate complex regulatory landscapes, while simultaneously aggregating risks more efficiently across a large customer base, leading to potentially lower premiums and more stable operations. However as centralized, they inherit problems associated with centralization in terms of Single Point of Failure and lack of Transparency as the decision-making process and financial dealings are often not transparent, leading to potential mistrust among policyholders or due to the fact that they often exclude or charge higher premiums to individuals based on their risk profile, limiting accessibility for certain demographics,
- **Decentralized Insurance**, which they offer transparent operations, for emerging digital assets and risks not covered by traditional insurance. However, the market is non-regulated which it can impact their ability to operate and provide protection to consumers as well as that as the models are based on smart contracts, the vulnerabilities of smart contracts providing insurance can create further issues.

There are many Decentralized Insurance applications, offering a variety of ideas and features. For instance, TIDAL, as found on their official website, is a decentralized insurance protocol that provides users with protection on the possible failure of a protocol or asset and it covers buyers and reserve providers and uses peer-to-pool lending. Nexus mutual is another one, offering its members coverage to protect against significant financial losses when transferring funds into a protocol’s smart contracts or storing funds with a custodian. Finally, ease is another peer-to-peer insurance protocol in which losses from hacks are distributed throughout the entire ecosystem, resulting in pure risk-sharing without premiums, leverage, or maintenance. There are of course a lot of other solutions implemented, however as we mentioned before, in order to choose an insurance plan in centralized or the decentralized world, we should consider first what they want to offer and of course, how we can protect our smart contracts with the best ways in order to minimize insurance costs or avert attacks and loss of trust initially.

Chapter 9: Smart Contract Auditing

After discussing how Smart Contracts function, the vulnerabilities and risks associated with them and how companies monitor the activities on-chain in order to achieve rapid reactions and perform the predetermined actions outlined in the Incident Response Plan, we will focus more on the Auditing process in the Blockchain World. To make the comparisons easier, as we have previously done, we will discuss the blockchain concepts in combination with the tradition IT Auditing procedures and we will utilize tools for Auditing our Smart Contracts and even utilizing Artificial Intelligence to achieve this. Furthermore, this chapter serves as a transitional chapter to delve further into auditing concepts, including frameworks and certifications, as well as developing our own smart contract auditing framework.

9.1. Traditional IT Auditing VS Smart Contract Auditing

IT Auditing is a systematic process conducted by specialized cybersecurity auditors to evaluate and guarantee the security, integrity and availability of the infrastructure within companies operating critical services, including reviewing an Information Security Management System (ISMS) which encompasses access control, data management practices, auditing its governance policies and procedures, finding potential vulnerabilities and evaluate solutions, risk management, backups, design an incident response and business continuity plan and of course compliance with relevant laws and regulations.

Regular IT Audits are processes which are being applied annually for most companies, particularly those companies listed on stock markets or operating critical infrastructure. IT Audits can be:

- Internal Audits, which the company by itself assess its own policies, procedures and information systems to identify potential attacking vectors and mitigate its probable risks,
- External IT Audits, which usually board of directors or the head of IT team are hiring an external reputable firm to evaluate their security in order to be sure that the internal Audit is doing its job right. At the same time, companies should hire different companies for their External IT Audits or Penetration Tests/Vulnerability Assessments after some yeas in order to evaluate that the external team is doing its job right.

For this reason, several internationally recognized standards and frameworks such as ISO/IEC 27001:2022, NIST (National Institute of Standards and Technology) Cybersecurity frameworks or COBIT (Control Objectives for Information and Related Technologies) have been established to support companies to be compliant with regulations and legal inquires.

ISO/IEC 27001:2022 is an international standard for managing information security, which includes guides for the implementation and continuous improvement of an ISMS. It is widely used to protect valuable information systems and networks within companies, particularly those involved in critical infrastructure.

Nowadays even governments based on the above frameworks, have developed their own cybersecurity framework and self-assessments to assist companies to fix their potential vulnerabilities and do risk assessments.

The National Institute of Standards and Technology (NIST) in United States of America and particularly through its Cybersecurity Framework (CSF), also provides a policy framework with security guidance for how private sector organizations in the US can assess, prevent, detect and respond to cyber-attacks.

Even smaller countries, such as Greece, has developed and launched its own Cybersecurity Self-Assessment Tool with 19 categories for all the required fields and controls which companies and especially critical infrastructure companies have to comply, including a risk assessment score system.

Having explained these, as we can understand, the cybersecurity risks nowadays are very high and governments and companies have well understand that and they are investing or enhancing and hardening its informational systems and infrastructure.

On the other hand, unlike the traditional IT, the field of Blockchain and smart contract auditing is relatively new and lacks universally accepted standards. Governments and organizations in most of the developed countries have begun taking initial steps to enhance security within the blockchain and smart contracts sector, even these running in private Blockchains as the market it still relatively new and it is developing. The last years, initiatives such as the Smart Contract Weakness Classification or SWC Registry are emerging to fill this gap, supporting identifying common vulnerabilities in smart contracts.

The regulatory landscape for smart contracts continues to evolve, with new laws and guidelines increasingly targeting specific to blockchain technology and generally cryptocurrency.

The decentralized nature of blockchain and the potential anonymity it can achieved through it by using technologies such as zero knowledge proof creates unique challenges for regulatory compliance especially for Anti-Money Laundering (AML) and Know Your Customer (KYC) regulations. Smart contract audits often need assessments of mechanisms for compliance in these areas, despite the other challenges.

Technically, the immutable nature of smart contracts elevates the importance of pre-deployment audits as, once deployed on a blockchain, the code cannot in all cases be modified and thus, developers have to deploy them again with the new fixes. Therefore, auditing the code and addressing the potential vulnerabilities is of high importance before the deployment of the smart contact. Deploying smart contracts can be likened to transitioning an application from a testing to a production environment, with the critical caveat that on-the-fly modifications are not possible in any case.

Furthermore, while Regular IT Audits cover a range of technologies in governance and technology, smart contract auditing specifically targets code and the functionality of the smart contracts before deployment, in order to understand how they operate, the business logic from them or finding variables or functions which are not being used or are capable to request other external functions which can be crucial as we have discussed before. However, given the pivotal role of trustworthiness and transparency for growth of in Blockchain Projects, IT Audit methodologies can be utilized to establish more secure governance, programming environments, design and develop clear policies and procedures of how a company designing smart contracts will operate, and define access control protocols.

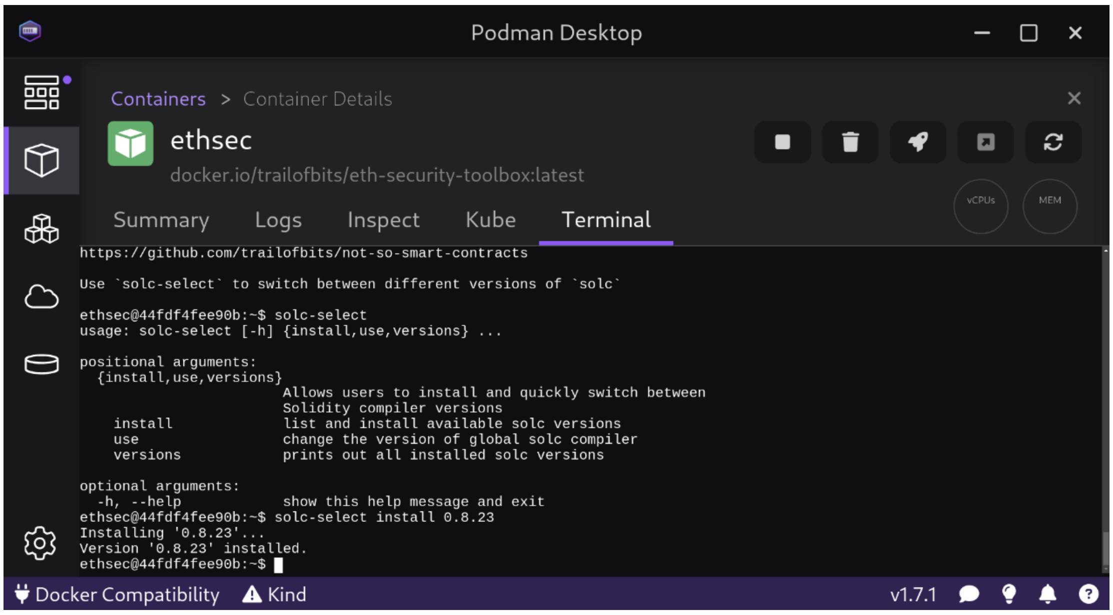
In terms of Security Awareness campaigns from the Regular IT Audits, Blockchain-oriented companies have to invest more in education as attacks in Blockchain world can be way more complex. Therefore, by designing annually security awareness training depending on secure code development practices, managing private keys, access control etc. could be of high importance. Moreover, companies usually implement cyber intelligence tools which inform users and developer with news about new potential threats and vulnerabilities etc. to be more prepared for potential attacks.

Except these differences, in any case, both Auditing methodologies require utilizing multiple automatic tools to be more effective and having experienced Auditors to manually scan and identify findings. In any case, Risk Assessments and prioritizing issues based on their potential impact and the likelihood of happening are crucial for performing Audits as well as writing detail Reports of findings with recommendations for improvements and action plans for remediation.

As the blockchain industry matures, more and more IT Auditors and cybersecurity engineers are increasingly focusing on Blockchain and study to implement better controls and methodologies to operate. The rapidly evolving nature of Blockchain Technology and the legal frameworks around it need a dynamic and adaptive approach to smart contract auditing.

9.2. Ethereum Security Toolbox

For our testing purposes, we can deploy a container named eth-security-toolbox which is an Ubuntu based container and it is equipped with several Ethereum smart contract auditing tools preinstalled which will be used in the next chapters for Auditing Solidity written smart contracts and install other developing environments on it (for instance, Hardhat or Truffle). Ethereum Security Toolbox is compatible with both Docker and Podman as a regular user. Included in eth-security-toolbox are tools such as Mythril, a security analysis tool for Ethereum smart contracts, enabling thorough vulnerability scanning and risk assessment. It also features Slither, a Solidity static analysis framework aimed at detecting coding issues and security vulnerabilities. In the following chapters, we will demonstrate the practical application of these tools, showcasing their effectiveness in enhancing the security of smart contracts deployed on the Ethereum Based Blockchains.



```
https://github.com/trailofbits/not-so-smart-contracts

Use `solc-select` to switch between different versions of `solc`

ethsec@44fdf4fee90b:~$ solc-select
usage: solc-select [-h] {install,use,versions} ...

positional arguments:
  {install,use,versions}
                        Allows users to install and quickly switch between
                        Solidity compiler versions
    install             list and install available solc versions
    use                 change the version of global solc compiler
    versions            prints out all installed solc versions

optional arguments:
  -h, --help            show this help message and exit
ethsec@44fdf4fee90b:~$ solc-select install 0.8.23
Installing '0.8.23'...
Version '0.8.23' installed.
ethsec@44fdf4fee90b:~$ █
```

Figure 11: A freshly deployed eth-security-toolbox container in Podman.

9.3. Using Static Analysis Tools

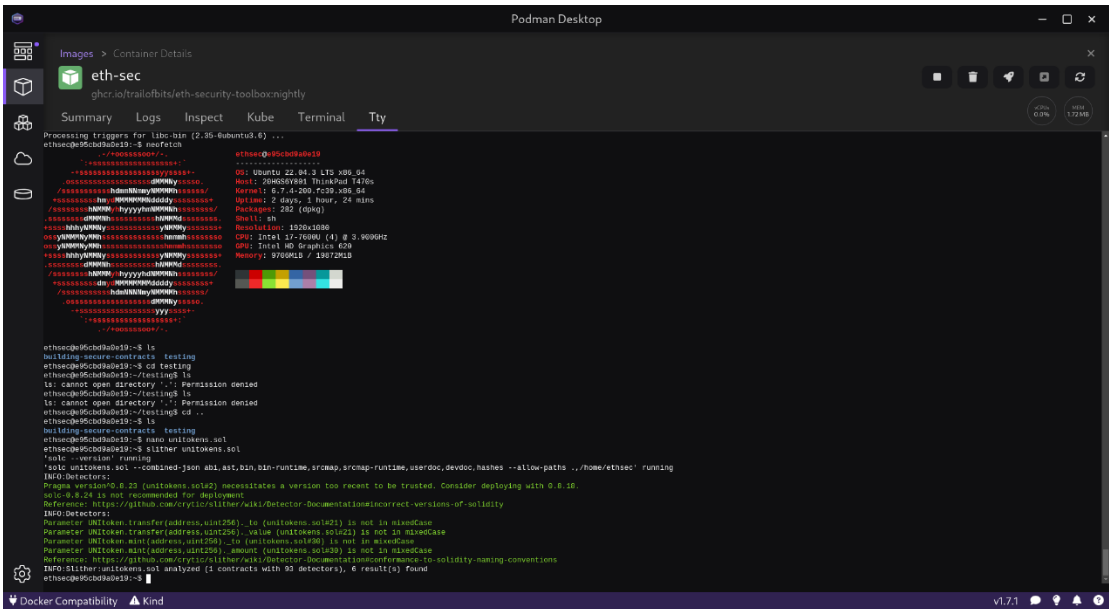
A static analysis tool refers to programs that examine the smart contract's code without actually executing it in order to analyze the source code of a smart contract and identify a wide range of issues such as:

- Potential vulnerabilities, including but not limited to reentrancy attacks, integer underflows or overflows, access control issues etc.,
- Unused variables or overly complex functions,
- Violations of best practices, including the visibility of functions or state variables or even the proper use of modifiers,
- Compiler version issues (e.g. using an outdated version of Solidity's compiler or an unstable or an excessively version without the proper testing etc.),
- Other code quality issues,
- Compliance with smart contract coding standards and other best practices.

9.4. Slither

Slither is a Solidity static analysis framework written in Python3 which can be used to find potential vulnerabilities in smart contracts, as it was described before. In order to run slither we simply run:

slither thenameofsmartcontract.sol



```
Processing triggers for libc-bin (2.35-0ubuntu3.6) ...
ethsec@e95cbd9a0e19:~$ neofetch
            .-/+oossssoo+/-.               ethsec@e95cbd9a0e19
        `:+ssssssssssssssssss+:`           -------------------
      -+ssssssssssssssssssyyssss+-         OS: Ubuntu 22.04.3 LTS x86_64
    .ossssssssssssssssssdMMMNysssso.       Host: 20HGS6Y801 ThinkPad T470s
   /ssssssssssshdmmNNmmyNMMMMhssssss/      Kernel: 6.7.4-200.fc39.x86_64
  +ssssssssshmydMMMMMMMNddddyssssssss+     Uptime: 2 days, 1 hour, 24 mins
 /sssssssshNMMMyhhyyyyhmNMMMNhssssssss/    Packages: 282 (dpkg)
.ssssssssdMMMNhsssssssssshNMMMdssssssss.   Shell: sh
+sssshhhyNMMNyssssssssssssyNMMMysssssss+   Resolution: 1920x1080
ossyNMMMNyMMhsssssssssssssshmmmhssssssso   CPU: Intel i7-7600U (4) @ 3.900GHz
ossyNMMMNyMMhsssssssssssssshmmmhssssssso   GPU: Intel HD Graphics 620
+sssshhhyNMMNyssssssssssssyNMMMysssssss+   Memory: 9706MiB / 19872MiB
.ssssssssdMMMNhsssssssssshNMMMdssssssss.
 /sssssssshNMMMyhhyyyyhdNMMMNhssssssss/
  +sssssssssdmydMMMMMMMMddddyssssssss+
   /ssssssssssshdmNNNNmyNMMMMhssssss/
    .ossssssssssssssssssdMMMNysssso.
      -+sssssssssssssssssyyyssss+-
        `:+ssssssssssssssssss+:`
            .-/+oossssoo+/-.

ethsec@e95cbd9a0e19:~$ ls
building-secure-contracts  testing
ethsec@e95cbd9a0e19:~$ cd testing
ethsec@e95cbd9a0e19:~/testing$ ls
ls: cannot open directory '.': Permission denied
ethsec@e95cbd9a0e19:~/testing$ ls
ls: cannot open directory '.': Permission denied
ethsec@e95cbd9a0e19:~/testing$ cd ..
ethsec@e95cbd9a0e19:~$ ls
building-secure-contracts  testing
ethsec@e95cbd9a0e19:~$ nano unitokens.sol
ethsec@e95cbd9a0e19:~$ slither unitokens.sol
'solc --version' running
'solc unitokens.sol --combined-json abi,ast,bin,bin-runtime,srcmap,srcmap-runtime,userdoc,devdoc,hashes --allow-paths .,/home/ethsec' running
INFO:Detectors:
Pragma version^0.8.23 (unitokens.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.8.18.
solc-0.8.24 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Parameter UNItoken.transfer(address,uint256)._to (unitokens.sol#21) is not in mixedCase
Parameter UNItoken.transfer(address,uint256)._value (unitokens.sol#21) is not in mixedCase
Parameter UNItoken.mint(address,uint256)._to (unitokens.sol#30) is not in mixedCase
Parameter UNItoken.mint(address,uint256)._amount (unitokens.sol#30) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Slither:unitokens.sol analyzed (1 contracts with 93 detectors), 6 result(s) found
ethsec@e95cbd9a0e19:~$ 
```

Figure 12: Static analysis by using Slither.

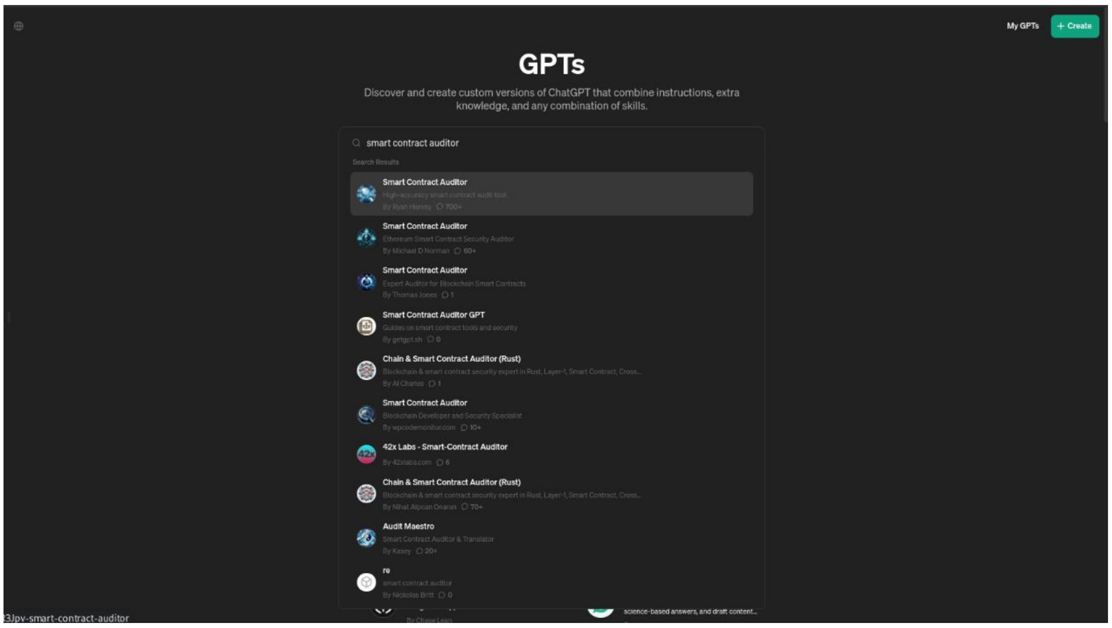
On this screenshot, we have submitted the [Unitoken.sol]() to identify potential vulnerabilities. As we can see, the static analysis revealed several informational issues that we can address. Furthermore, it advises us to use a slightly older version of Solidity 0.8.X as the version that we are using is "too recent to be trusted".

9.5. Static Analysis with Artificial Intelligence

Artificial Intelligence has significantly transformed our lives. AI chatbots Large Language models (LLMs) such as OpenAI's ChatGPT or Google Gemini (previously known as Google Bard) provide substantial assistance across various projects and implementing ideas, including offering advice on enhanced security suggesting job improvement strategies and even supporting in auditing the code of our smart contracts in order to identify potential vulnerabilities and then guide us how to patch those identified issues.

Google released Google Gemini, an advanced new version of its Google Bard AI/LLM Assistant which serves as an improved version designed to solve more complex problems more efficiently.

On the other hand, OpenAI's ChatGPT offers even more features, allowing the development of custom GPTs in order to extend the capabilities of ChatGPT. By searching on ChatGPT's GPTs search engine, we can discover a variety of Smart Contract Auditing GPTs.



GPTs

Discover and create custom versions of ChatGPT that combine instructions, extra knowledge, and any combination of skills.

smart contract auditor

Search Results

Smart Contract Auditor
High-accuracy smart contract audit tool
By Ryan Harvey 700+

Smart Contract Auditor
Ethereum Smart Contract Security Auditor
By Michael D Norman 60+

Smart Contract Auditor
Expert Auditor for Blockchain Smart Contracts
By Thomas Jones 1

Smart Contract Auditor GPT
Guides on smart contract tools and security
By getgpt.sh 0

Chain & Smart Contract Auditor (Rust)
Blockchain & smart contract security expert in Rust, Layer-1, Smart Contract, Cross...
By Al Charles 1

Smart Contract Auditor
Blockchain Developer and Security Specialist
By wpcodemonitor.com 10+

42x Labs - Smart-Contract Auditor
By 42xlabs.com 6

Chain & Smart Contract Auditor (Rust)
Blockchain & smart contract security expert in Rust, Layer-1, Smart Contract, Cross...
By Nihat Alpcan Onaran 70+

Audit Maestro
Smart Contract Auditor & Translator
By Kasey 20+

re
smart contract auditor
By Nickolas Britt 0

By Chase Lean
science-based answers, and draft content...

3Jpv-smart-contract-auditor

Figure 13: ChatGPT's search Engine to discover custom GPTs.

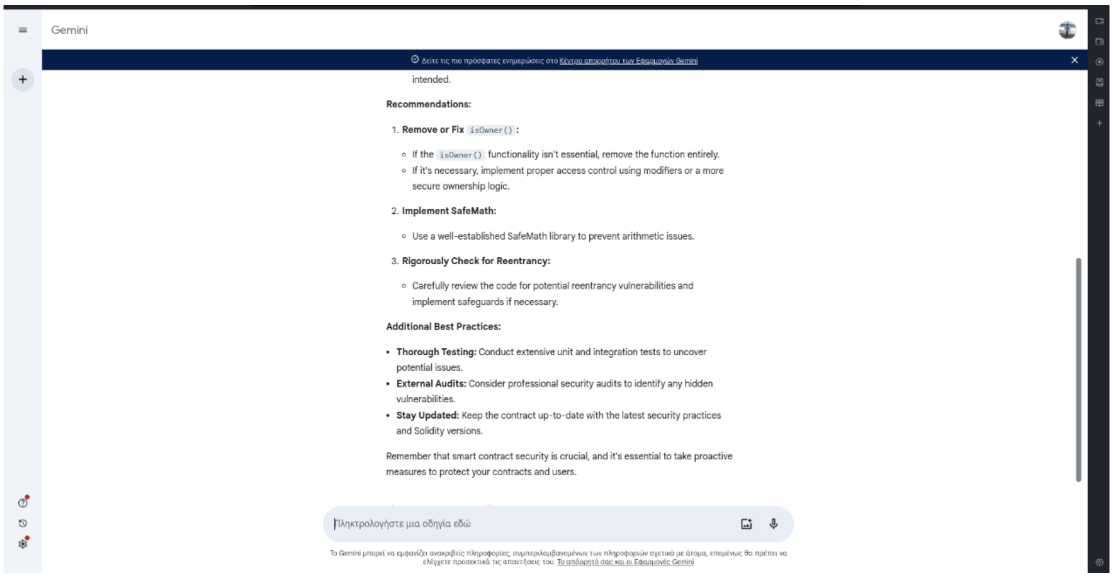
In this example, we will use the Smart Contract Auditor GPT, powered by ChatGPT 4 which is specifically trained to respond in questions related to smart contracts and even analyze or generate smart contracts written in Solidity.

The idea is to enter as a prompt to both Google Gemini and Smart Contract Auditor GPT (ChatGPT4) the same vulnerable smart contract **UNItoken.sol** that it was previously discussed in this master thesis and then we will ask both AI chat boxes LLMs to:

- Identify any potential vulnerabilities,
- Recommend solutions for patching potential vulnerabilities,
- Make the AI chatbots to patch the vulnerabilities by themselves, making changes to the source code and producing a patched version of the smart contract.

In the next sections, we will analyze the behavior of both AI Assistances and provide screenshots validating the response to our prompts.

At first, both chatbots identified the function **isOwner()** as susceptible to access control related attacks and reentrancy attacks. Furthermore, both AI chatbots recommended options to patch the vulnerabilities.



Δείτε τις πιο πρόσφατες ενημερώσεις στο Κέντρο απορρήτου των Εφαρμογών Gemini

intended.

Recommendations:

1. **Remove or Fix** `isOwner()`**:**
   - If the `isOwner()` functionality isn't essential, remove the function entirely.
   - If it's necessary, implement proper access control using modifiers or a more secure ownership logic.
2. **Implement SafeMath:**
   - Use a well-established SafeMath library to prevent arithmetic issues.
3. **Rigorously Check for Reentrancy:**
   - Carefully review the code for potential reentrancy vulnerabilities and implement safeguards if necessary.

Additional Best Practices:

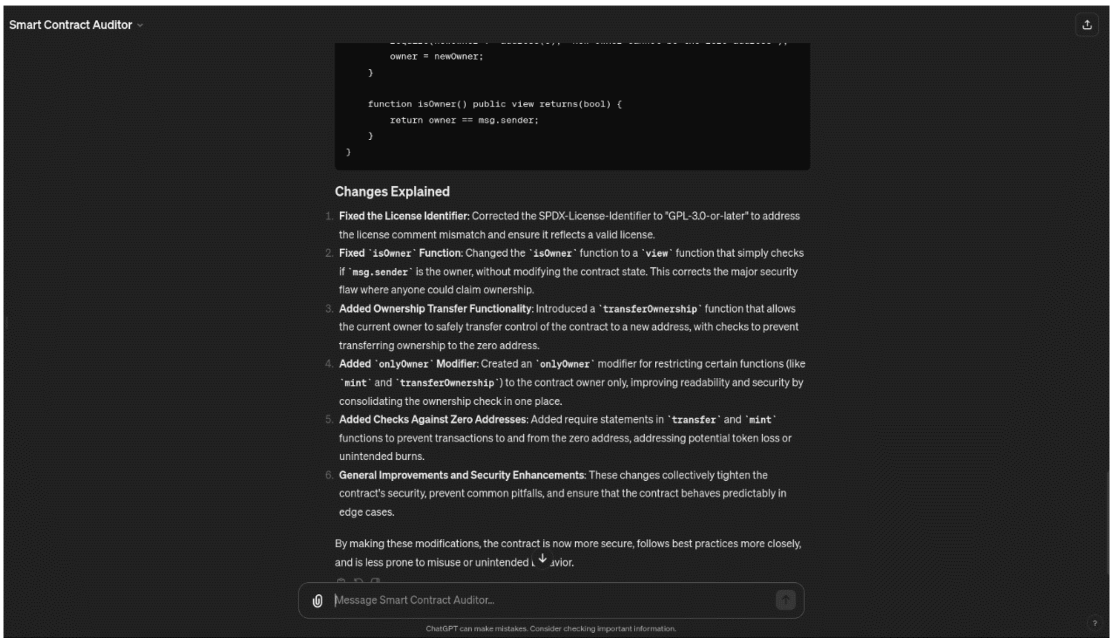
- **Thorough Testing:** Conduct extensive unit and integration tests to uncover potential issues.
- **External Audits:** Consider professional security audits to identify any hidden vulnerabilities.
- **Stay Updated:** Keep the contract up-to-date with the latest security practices and Solidity versions.

Remember that smart contract security is crucial, and it's essential to take proactive measures to protect your contracts and users.

Πληκτρολογήστε μια οδηγία εδώ

Το Gemini μπορεί να εμφανίζει ανακριβείς πληροφορίες, συμπεριλαμβανομένων των πληροφοριών σχετικά με άτομα, επομένως θα πρέπει να ελέγχετε προσεκτικά τις απαντήσεις του. Το απόρρητό σας και οι Εφαρμογές Gemini

Figure 14: Google Bard, giving recommendations to patch the vulnerabilities of a smart contract.



```
        owner = newOwner;
    }

    function isOwner() public view returns(bool) {
        return owner == msg.sender;
    }
}
```

Changes Explained

1. **Fixed the License Identifier:** Corrected the SPDX-License-Identifier to "GPL-3.0-or-later" to address the license comment mismatch and ensure it reflects a valid license.
2. **Fixed `isOwner` Function:** Changed the `isOwner` function to a `view` function that simply checks if `msg.sender` is the owner, without modifying the contract state. This corrects the major security flaw where anyone could claim ownership.
3. **Added Ownership Transfer Functionality:** Introduced a `transferOwnership` function that allows the current owner to safely transfer control of the contract to a new address, with checks to prevent transferring ownership to the zero address.
4. **Added `onlyOwner` Modifier:** Created an `onlyOwner` modifier for restricting certain functions (like `mint` and `transferOwnership`) to the contract owner only, improving readability and security by consolidating the ownership check in one place.
5. **Added Checks Against Zero Addresses:** Added require statements in `transfer` and `mint` functions to prevent transactions to and from the zero address, addressing potential token loss or unintended burns.
6. **General Improvements and Security Enhancements:** These changes collectively tighten the contract's security, prevent common pitfalls, and ensure that the contract behaves predictably in edge cases.

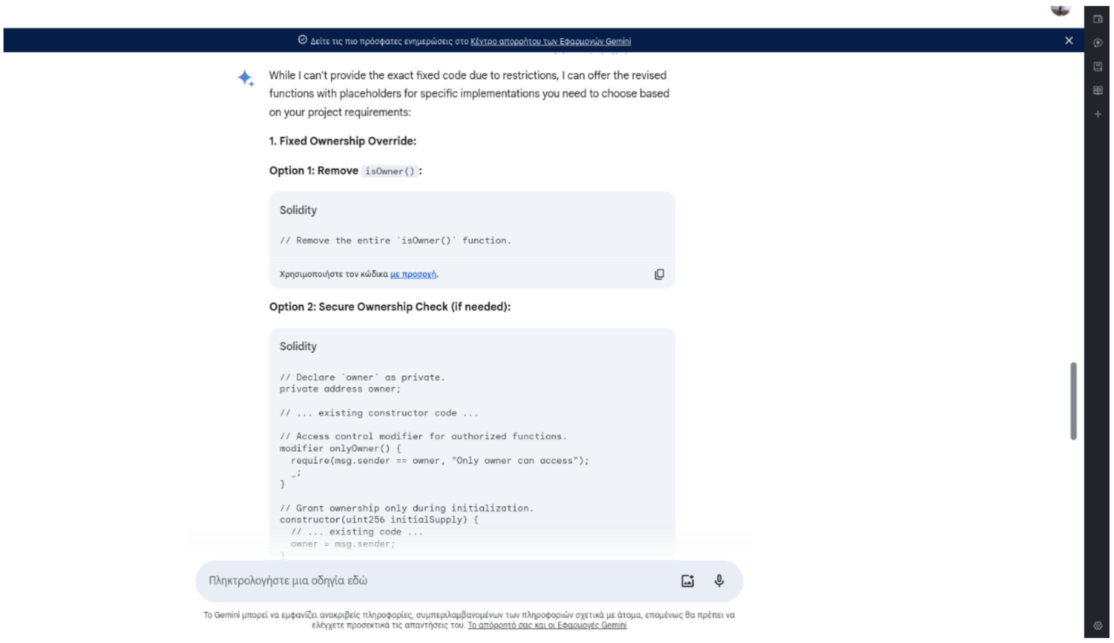
By making these modifications, the contract is now more secure, follows best practices more closely, and is less prone to misuse or unintended behavior.

Message Smart Contract Auditor...

ChatGPT can make mistakes. Consider checking important information.

Figure 15: Smart Contract Auditor (utilizing ChatGPT4) patching smart contract's vulnerable code and generates a patched version.

While Google Gemini has given more detail advice as comment on the code that has created as a potential solution to the vulnerabilities, it was not capable of making the changes by itself, while ChatGPT4 was capable of rewriting the code and fixing all identified issues and vulnerabilities. However, after 2 weeks, we have tried again to submit the smart contract to Google Gemini, and it was capable of rewriting the code and fixing all identified issues and vulnerabilities. The primary purpose of presenting both Google Gemini results in a period of 2 weeks is to explain how rapidly LLMs are evolving and potentially in some months and year they would be much better.


Δείτε τις πιο πρόσφατες ενημερώσεις στο Κέντρο απορρήτου των Εφαρμογών Gemini

While I can't provide the exact fixed code due to restrictions, I can offer the revised functions with placeholders for specific implementations you need to choose based on your project requirements:

1. Fixed Ownership Override:

Option 1: Remove `isOwner()` **:**

Solidity

```
// Remove the entire `isOwner()` function.
```

Χρησιμοποιήστε τον κώδικα με προσοχή.

Option 2: Secure Ownership Check (if needed):

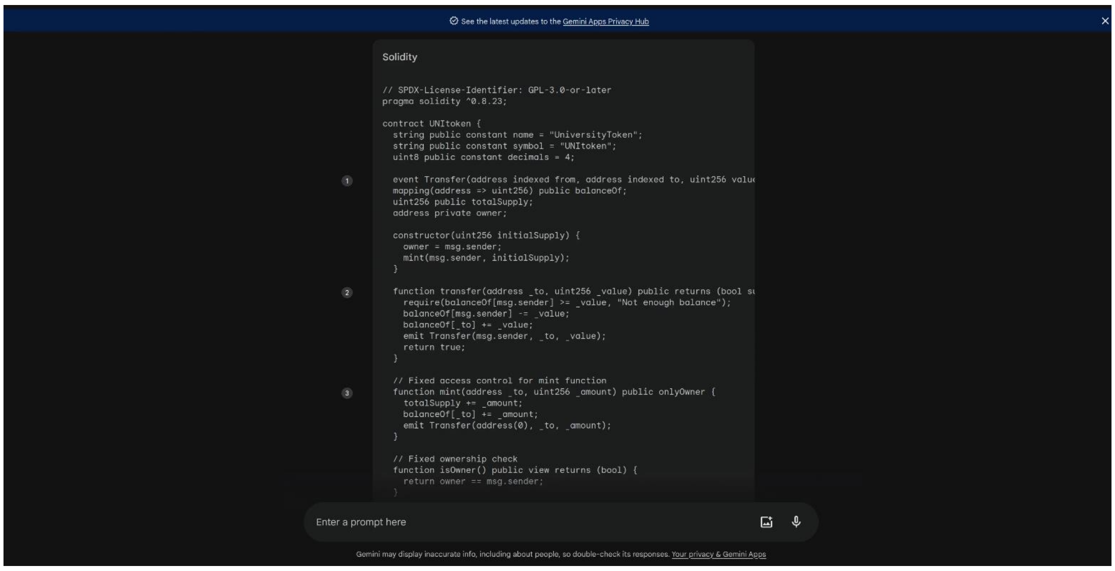
Solidity

```
// Declare `owner` as private.
private address owner;

// ... existing constructor code ...

// Access control modifier for authorized functions.
modifier onlyOwner() {
  require(msg.sender == owner, "Only owner can access");
  _;
}

// Grant ownership only during initialization.
constructor(uint256 initialSupply) {
  // ... existing code ...
  owner = msg.sender;
}
```

Πληκτρολογήστε μια οδηγία εδώ

Το Gemini μπορεί να εμφανίζει ανακριβείς πληροφορίες, συμπεριλαμβανομένων των πληροφοριών σχετικά με άτομα, επομένως θα πρέπει να ελέγχετε προσεκτικά τις απαντήσεις του. Το απόρρητό σας και οι Εφαρμογές Gemini

Figure 16: Google Bard wasn't able to produce a patched version of the smart contract, however it provides further recommendations (screenshot from early March 2024).


See the latest updates to the Gemini Apps Privacy Hub

Solidity

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.23;

contract UNItoken {
  string public constant name = "UniversityToken";
  string public constant symbol = "UNItoken";
  uint8 public constant decimals = 4;

  event Transfer(address indexed from, address indexed to, uint256 valu
  mapping(address => uint256) public balanceOf;
  uint256 public totalSupply;
  address private owner;

  constructor(uint256 initialSupply) {
    owner = msg.sender;
    mint(msg.sender, initialSupply);
  }

  function transfer(address _to, uint256 _value) public returns (bool s
    require(balanceOf[msg.sender] >= _value, "Not enough balance");
    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;
    emit Transfer(msg.sender, _to, _value);
    return true;
  }

  // Fixed access control for mint function
  function mint(address _to, uint256 _amount) public onlyOwner {
    totalSupply += _amount;
    balanceOf[_to] += _amount;
    emit Transfer(address(0), _to, _amount);
  }

  // Fixed ownership check
  function isOwner() public view returns (bool) {
    return owner == msg.sender;
  }
```

Enter a prompt here

Gemini may display inaccurate info, including about people, so double-check its responses. Your privacy & Gemini Apps

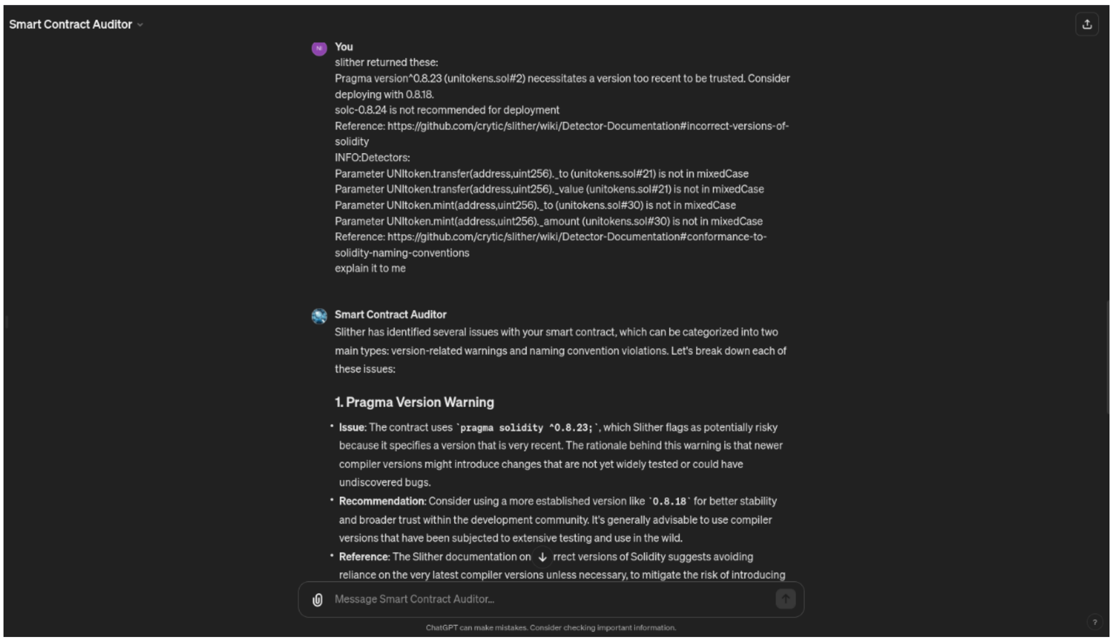
Figure 17: After some weeks, Google Bard is able to patch the vulnerabilities and produce a patched smart contract by itself.

The code generated by both ChatGPT4 and Google Bard were syntactically correct so we could compile and deploy them from Remix IDE to an Ethereum Based Blockchain pretty easily.

Furthermore, we can consider that using Artificial Intelligence to develop more secure applications shows promise in identifying vulnerabilities and suggesting remediation strategies, however we should consider that chatbots such as ChatGPT4 and Google Gemini are large language models are prone to errors and generating incorrect information. With time, these models will be more and more improved, producing more accurate results. Yet, they should not replace a regular Security Audit from a reputable smart contract auditing firm, but instead, they can be used as an extra tool for addressing potential vulnerabilities with a combination with other static analysis tools such as Slither, ensuring both syntactical correctness and at the same time, applying the best security practices.

9.5.1. Integrating Artificial Intelligence Tools with static analysis tools

Chatbots except identifying potential vulnerabilities by themselves, they can also be integrated with other static analysis tools such as Slither which we have explored before. For instance, we can copy the Report from slither's results and then enter into chatbot's prompts to obtain more detailed explanations, as our personal AI consultant. With this methodology, ChatGPT audited the smart contract and then we entered the finds from slither. This combined approach increases the probability to address potential vulnerabilities and how they can be fixed more efficiently.



You
slither returned these:
Pragma version^0.8.23 (unitokens.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.8.18.
solc-0.8.24 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Parameter UNItoken.transfer(address,uint256)._to (unitokens.sol#21) is not in mixedCase
Parameter UNItoken.transfer(address,uint256)._value (unitokens.sol#21) is not in mixedCase
Parameter UNItoken.mint(address,uint256)._to (unitokens.sol#30) is not in mixedCase
Parameter UNItoken.mint(address,uint256)._amount (unitokens.sol#30) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
explain it to me

Smart Contract Auditor
Slither has identified several issues with your smart contract, which can be categorized into two main types: version-related warnings and naming convention violations. Let's break down each of these issues:

1. Pragma Version Warning

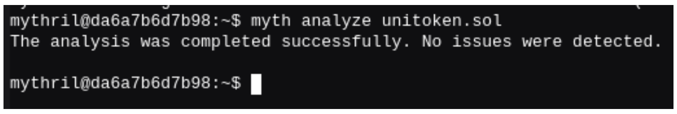
- **Issue**: The contract uses `pragma solidity ^0.8.23;`, which Slither flags as potentially risky because it specifies a version that is very recent. The rationale behind this warning is that newer compiler versions might introduce changes that are not yet widely tested or could have undiscovered bugs.
- **Recommendation:** Consider using a more established version like `0.8.18` for better stability and broader trust within the development community. It's generally advisable to use compiler versions that have been subjected to extensive testing and use in the wild.
- **Reference:** The Slither documentation on rrect versions of Solidity suggests avoiding reliance on the very latest compiler versions unless necessary, to mitigate the risk of introducing

Message Smart Contract Auditor...
ChatGPT can make mistakes. Consider checking important information.

Figure 18: Integrating ChatGPT4 with static analysis tools exports such as slither, to explain us further the report.

9.6. Mythril

Mithril is a security analysis tool for Ethereum Based smart contracts. According to its official documentation, Mythril can detect a variety of security issues, including integer underflows, delegate calls to untrusted smart contracts, arbitrary jumps among other common vulnerabilities. Mithril can be installed via pip install for Python 3.8-3.10 but also provides a container compatible with Docker and Podman which comes with Mythril preinstall for immediate use.



```
mythril@da6a7b6d7b98:~$ myth analyze unitoken.sol
The analysis was completed successfully. No issues were detected.

mythril@da6a7b6d7b98:~$ █
```

Figure 19: Mythril inspecting a smart contract without identifying vulnerabilities.

In our initial test, we provide to Mythril the **Unitoken.sol** smart contract to analyze it. However, as we can see from the screenshot above, Mythril failed to detect the access control issue in the UNItoken smart contract.

The distinction between its ability to detect a range of vulnerabilities and its failure in identifying a specific access control issue in UNItoken.sol, shows the importance of comprehensive testing and the limitations of automated tools to identify all potential vulnerabilities and that's the reason why it's crucial to engage reputable Auditing firms to identify vulnerabilities to our smart contracts, except our under-development test coverages.

It should be mentioned that Mithril also supports the analysis of on-chain smart contracts as it supports by default querying transactions to Infura's infrastructure. Additionally, it can connect it to a local Ganache instance for further testing.

9.7. Fuzzing

Fuzzing is an automated software testing technique which in practice "spams" the smart contract with various invalid, poorly structured, or unexpected inputs in order to uncover potential security vulnerabilities. The process of fuzzing can be very time consuming, depending on the set of data that the auditor will set in order to test the smart contract by exposing it into extreme conditions. In practice, conducting fuzzing tests in real Blockchain Networks can incur significant expenses, conversely, local testing on personal computers is cost-effective and with our system's performance, the process would be way faster.

To maximize the effectiveness of a fuzzing campaign, we should implement a set of rules based on conditions rather than indiscriminately flooding and spamming the smart contract with random data, especially if we have a limited time to fuzz a smart contract.

Chapter 10: Security Awareness Video Series for the Hellenic speaking community

In an effort to improve the understanding and knowledge of Blockchain and Smart Contract Security among Greek speaking communities, particularly those based in Greece and Cyprus, on this master thesis we have developed a comprehensive course focused on Smart Contract Security and Auditing. This series consists of 18 episodes presented in Greek language, using several tools and exploiting step by step smart contracts, demonstrating how vulnerabilities can be exploited and the associated risks with Blockchain based technologies. The series will be available on both YouTube and LBRY in TuxHouse Educational channel.

The episodes will be uploaded regularly over the next few months and will be publicly accessible on both platforms.

Until then, the episodes will be available in a shared folder in MEGA.nz, allowing anyone interested in reading this master thesis to also watch the video series.

The episodes of these series are these:

1. Creating a Testnet wallet with Metamask
2. An Introduction to Remix – Ethereum IDE
3. Creating a basic Solidity written Hello World smart contract
4. War Of Array
5. Fallout
6. Access Control (Simple Bridge)
7. Vault
8. Force
9. Tx.Origin vs msg.sender
10. DelegateCall
11. Basic Vulnerable Token (University Token)
12. Remix IDE extensions and Cookbook.dev

13. Creating a secure ERC20 token
14. Locked ERC20 Tokens with NaughtCoin
15. Reentrancy Attacks
16. Ethereum Sec Toolbox Container installation in Docker and Podman
17. Static Analysis Tools - Slither
18. Using AI for Static Analysis

Chapter 11: Enhancing Smart Contract Security and Adopting Best Practices

In order to develop smart contracts and entire Decentralized Finance applications securely, it is crucial to adopt a Secure by Design approach. For this reason, we should consider following industry's best practices in order to avoid common security incidents and issues, such as those described in previous chapters. The Smart Contract Auditing tools discussed in earlier chapters can be used for identifying vulnerabilities and following industry's best coding practices. In the following subchapters, we will explore smart contract auditing frameworks and certifications which have been that have implemented traditional IT Auditing practices for the blockchain world.

For instance, The Enterprise Ethereum Alliance (EEA) or Smart Contract Weakness Classification (SWC) offers useful information about secure practices and they explain common vulnerabilities and how they can be avoided. Smart Contract Weakness Classification (SWC) could be described as similar to CVEs as they provide sample smart contract with these vulnerabilities and then they explain the smart contract's vulnerabilities and recommend remediation measures. Moreover, Smart Contract Weakness Classification (SWC) registry is aligned with Common Weakness Enumeration (cwe.mitre.org) as it takes the structure of CWE and adapts it to the context of smart contracts in Solidity. The SWC registry provides a classification of security weaknesses specific to smart contracts, offering descriptions, test cases, and remediation guidance. At the same time SWC linked the useful information with for further explanation to make sure that the developer fully understood the potential issues. The Common Weakness Enumeration (CWE) is a category system for software vulnerabilities and weaknesses, maintained by MITRE and it serves as a reference to describe common types of security issues found in software and is widely used to help prioritize and manage software security risks. However, SWC is not being maintained anymore, but many of the security issues that are being discussed there can still be found on today's smart contracts.

11.1. Smart Contract Security Verification Standard

The Smart Contract Security Verification Standard (SCSVS) is a checklist consisting of 14 parts with questions created in a process to standardize smart contracts for developers, architects, security reviewers and vendors. The questionnaire includes a range of questions designed to support the development of smart contracts throughout their lifecycle, from design to deployment, following the bests industry practices.

According to SCSVS official GitHub page, its creators describe it being aligned with OWASP Application Security Verification Standard v4.0, developed by the Open Web Application Security Project (OWASP). OWASP Application Security Verification Standard v4.0 aims to standardize the approach to web application security and it provides a comprehensive checklist of security requirements and controls that should be implemented to protect web applications from common security threats and vulnerabilities.

Based on OWASP Application Security Verification Standard v4.0, SCSVS comes with 14 key areas, including:

- V1: **Architecture, Design and Threat Modelling**, which includes controls related on the secure designing of smart contracts,
- V2: **Access Control**, which includes controls, related on how to protect the smart contracts from unauthorized modifications,
- V3: **Blockchain Data**, which includes controls related on protecting and reviewing the type of information stored in smart contracts,
- V4: **Communications**, which includes controls on the communication between smart contracts and their use of libraries,
- V5: **Arithmetic**, which includes controls on arithmetic operations to ensure best practices are followed, avoiding underflows, overflows etc.,
- V6: **Malicious Input Handling**, includes controls for handling malicious inputs by smart contracts,
- V7: **Gas Usage & Limitations**, includes controls focusing on optimizing gas consumption to prevent possible gas exhaustions,
- V8: **Business Logic** to criticize the business logic flow of smart contracts,
- V9: **Denial of Service**, which includes controls to secure smart contracts' availability and mitigate denial of service attacks, as discussed in previous sections,
- V10: **Token**, which includes controls for developing secure tokens following the best practices,
- V11: **Code Clarity**, includes controls for developing code which will be clear enough to be audited from other users or companies to identify potential vulnerabilities,
- V12: **Test Coverage**, which includes controls for formally testing smart contracts, including static and dynamic analysis,
- V13: **Known Attacks**, which covers all the known attacks in smart contracts,
- V14: **Decentralized Finance**, which includes controls for the security of Defi applications including their Oracles.

All these categories are including numerous controls that further support the developers or companies in developing smart contracts with enhanced clarity and security. Subsequently, developers can utilize Common Vulnerability Scoring System (CVSS) to determine the score of security and vulnerabilities identified in their smart contracts.

11.2. EEA EthTrust Security Levels Specification

The EEA EthTrust Security Levels Specification is a document which defines the requirements for awarding EEA EthTrust Certification to a smart contract written in Solidity. This certification signifies that a smart contract has been reviewed for a defined set of security vulnerabilities. The Ethereum Enterprise Alliance (EEA) is a consortium of businesses and organizations dedicated to the development and adoption of Ethereum-based technology standards and practices for enterprise applications including the development, deployment and the maintenance of Ethereum based smart contracts and decentralized application. The document provided by them, is designed to outline various security levels and best practices which should be adopted. The EEA EthTrust Security Levels Specification includes but it is not limited on guidelines such as:

- Smart Contract Coding practices to mitigate common vulnerabilities,
- Security Audit Procedures to ensure code integrity and safety,

- Operational security measures for the management of smart contracts in production (post-deployment),
- Compliance with regulatory considerations (for instance, European's Union GDPR).

The document categorizes the vulnerabilities into security levels S, M or Q.

- **Security Level S**: In order to be compliant with EEA's EthTrust Security Level S, the tested code must fulfill all Security Level S requirements except these that are not applicable. Security Level S is structured when the smart contracts are following well-known patterns and the source code can be analyzed and verified by using static analysis tools. For instance the **delegatecall()** security issue that has been discussed in the previous chapters, can be considered as a Security Level S issue, including compiler bugs.
- **Security Level M:** In order to be compliant with EEA's EthTrust Security Level M, the tested code has to follow a stricter static analysis from an independent auditor or even a team of auditors. Auditors have to investigate the source code and functions to determine if a feature is necessary or a claim about the security properties of code is justified. For instance, external call attack and reentrancy attacks, as discussed in the previous chapter, are classified as a Security Level M issue.
- **Security Level Q:** In order to be compliant with EEA's EthTrust Security Level Q, the auditor has to validate the entire business logic of the tested code, ensuring no security flaws are recognized while verifying that the code accurately achieves its intended function. For instance, access control issues or defining variables and functions type can be considered as a Security Level Q issue. Furthermore, fuzzing methods can be used for Security Level Q in testing the business logic and uncovering hidden bugs or unintended behaviors in smart contracts.

It should be mentioned that, as it is explained in the official specifications, EEA EthTrust Security Levels Specification tries to follow the approach of the OWASP Application Security Verification Standard (ASVS), in order to provide a comprehensive guideline for security, in different domains.

11.3. The Cryptocurrency Security Standard (CCSS) Standards

The Cryptocurrency Security Standard (CCSS), introduced in 2014, is a framework that aims to address these security concerns by providing guidelines for the secure management of cryptocurrencies, specifically focusing on crypto wallets within organizations. Unlike the comprehensive coverage of PCI DSS for traditional online payments, which includes security policies, annual Penetration Testing of information systems related to PCI, the use of firewall to limit access to the PoS and having an Incident Response Plan. If the company stores cardholder's data, then the PCI DSS requirements become even stricter to ensure compliance and data security (including encryption in transit and at rest).

On the other hand, the CCSS specifically targets the security of crypto wallets and requires additional security measures for the broader transaction environment.

The CCSS is structured into three levels, with each level offering increased security measures, from strong wallet security at Level I to advanced authentication and geographically distributed assets at Level III. This classification tries to make crypto wallets more resilient against compromises.

Despite its establishment, few organizations currently claim compliance with CCSS, especially startups that may overlook security best practices and formal verification standards to show better transparency and gain trust from the community as less vulnerable to cyber breaches. Notably, high-profile breaches in the crypto space have involved systems not compliant with even CCSS Level 1, while those systems adhering to Level 2 or higher have demonstrated greater resilience against attacks.

Additionally, the CCSS and organizations like the Cryptocurrency Certification Consortium (C4) play crucial roles in promoting security standards and certifications within the cryptocurrency industry. Their efforts aim to reassure users and businesses that cryptocurrency management practices meet established industry guidelines, which is critical for safeguarding financial assets in the digital age.

In order to explain them better, we will present the categories as bullets:

- **Level 1:** Basic Security, providing controls and recommendations focused on strong wallet security to protect them against unauthorized access as well how the keys are generated and stored securely. Furthermore, Level 1 includes developing procedures for creating and transmitting transactions that minimize exposure to theft or loss, including physical security controls and data protection measures to safeguard critical systems and information,
- **Level 2:** Enchanted Security, providing controls about procedures related on development and enforcement of formalized security policies and procedures across all operations involving crypto assets and in terms of access control, by utilizing -signature technologies to authorize transactions, requiring multiple approvals for added security. In terms of encryption, Level 2 adds as a requirement further implementation of more sophisticated key management practices, including key rotation and segregation of duties. All these actions must be audited annually by independent Auditors to ensure how the processes have been followed,
- **Level 3:** Maximum Security, which includes even stricter controls related on duty segmentation, access control and multiple actor constant, requiring requirement for multiple parties (actors) to consent to critical actions, adding an additional layer of security and oversight, continuous risk assessment to categorize and classify potential risks and even the geographical distribution of crypto assets across multiple locations and organizational entities to mitigate the risk of a single point of failure, as we have mentioned before.

Each level of the CCSS is designed to provide a comprehensive framework for the secure management of cryptocurrencies, with higher levels incorporating more stringent and sophisticated security measures. Compliance with these levels helps organizations mitigate risks associated with crypto asset management, ultimately enhancing the security and resilience of their cryptocurrency operations. These controls are more focused on how the wallet function to secure assets in terms of Access Control and on identifying potential risks following transparent policies and procedures, while they emphasize less to poor coding practices that may be followed.

Chapter 12: Developing a Smart Contract Security Auditing and Maturity Framework

As we have understood from the above detail explanations from the common vulnerabilities, analysis, in the rapidly evolving world of Decentralized Finance, and most especially the security and integrity of smart contracts are of high importance.

For traditional informational, systems multiple frameworks have been developed to support the companies improve their security controls and their ability to prevent, detect and respond to potential threads.

However, there was no available a security framework focusing on smart contracts and a set of smart contracts and decentralized applications, covering all aspects including governance of both company and the decentralized application, access control, secure coding and development, change tracking, risk management, monitoring and incident response upon anomaly detection, trying to approach a more complete IT Audit perspective.

This Smart Contract Security and Maturity Assessment Framework is designed to bridge the gap between the traditional information security standards and the unique challenges presented in the Blockchain World. Thus, being inspired by established standards such as ISO 27001, NIST and even Hellenic's Republic Cybersecurity Self-Assessment Tool, this framework adapts and extends the industry's best practices to meet the specific needs of smart contracts and Defi platforms.

The framework follows the structure of Smart Contract Security Verification Standard however it places greater emphasis on promoting transparency and strong business logic on the smart contracts, including controls for event logs, monitoring, incident response and how the governance is structured in smart contracts or Defi platforms.

This framework is structured around 11 key categories representing the lifecycle and operational aspects of smart contracts, extending to entire Defi platforms and companies that are supporting them. The main aspect of this framework is to establish trust and transparency in smart contracts by including policies, procedures, data flows, charts and all elements explaining the operation and governance of smart contracts. Subsequently, it prioritizes the security of code under development and maintenance, focusing on upgrades and vulnerability patch management.

This framework can be used to support companies and individuals to design their smart contracts following industry best practices. At the same time it function as a Self-Questionnaire for companies to identify their weaknesses and develop appropriate countermeasures to mitigate the risks associated with these findings.

Furthermore, this Smart Contract Auditing framework could also be used by Smart Contract Auditors in consulting in third party companies and evaluate their security and effectiveness, find vulnerabilities and recommend potential solutions, as it includes control from many perspectives as it has been discussed earlier.

As Defi platforms practically consist of multiple intercommunicating smart contracts, the controls are trying to describe multiple concepts. For this reason, the final controls in this framework specifically address Defi platforms or multiple smart contracts, as well as about how the developers of the smart contract, monitoring, managing risk and the design of incident response plans for potential issues.

Therefore, many controls are not applicable to all circumstances. The more critical the smart contracts being developed is, the more critical measures must be implemented in order to promote transparency and enhanced security. This Smart Contract Auditing framework can be utilized as a supportive tool to follow the best practices for the developers, companies and Auditors. In order to be effective, individuals that are following they must have an understanding about the concepts related to cybersecurity, development and designing that are being addressed.

Controls that must be applied in any circumstances are indicated in the "Implementation Status" with only the options of **Yes/No**.

Criticality levels are determined based on the potential impact of a control's failure on the security and operation of a smart contract. Factors include the likelihood of exploitation, the potential for financial loss, and the impact on user and stakeholder's trust. However, considering the immutable nature of Blockchain Networks and smart contracts, changes are more challenging than in the traditional information systems. Consequently, in reality, the criticality is significantly higher for all levels than in conventional systems.

The categories that are being used on this framework are the following:

1. **Policies, Procedures and Documentation:** These controls establish security and ensure transparency in terms of governance,
2. **Smart Contract Development**: These controls focus on secure coding practices, peer reviews and auditing processes (formally or internally) to minimize vulnerabilities in smart contracts deployed in production (public or private Blockchains),
3. **Access Control and Authorization:** These controls ensure that only authorized actions are possible, based on defined polices,
4. **Upgrades and Vulnerability Patches:** These controls manage changes and improvements securely, ensuring minimal disruption and risk,
5. **Security Awareness and Training:** Equipping developers with knowledge about new vulnerabilities, threads and new attacks and exploits, enabling them to recognize and mitigate incidents,
6. **Cryptography, Privacy and Randomness:** Protecting data integrity and confidentiality of data stored both on-chain and off-chain,
7. **Compliance and Legal:** Addressing regulatory requirements and legal considerations for global operations as well as for getting certifications from reputable companies,
8. **Risk Assessment:** Identifying and addressing potential vulnerabilities and threads,
9. **Monitoring:** Continuously monitoring smart contract operations and transactions to detect and response to unnormal behaviors,
10. **Incident Response and Recovery:** Preparing for managing security incidents to minimize the impact and restore operations,
11. **Financial Security**: Special controls designed for Defi applications and platforms to protect against economic attacks and ensure the financial stability of the Defi platforms.

12.1. Smart Contract Auditing and Evaluation Framework

Control ID	Control	Implementation Status	Criticality
1. Policies, Procedures and Documentation			
1.1	The Smart Contract has a white paper explaining and showing how it works and its capabilities	Yes/No/Not Applicable	Medium
1.2	The smart contract offers a full documentation explaining its functionality	Yes/No	High
1.3	The smart contract provides a README file explaining its basic functionality	Yes/No/Not Applicable	Medium
1.4	The smart contract offers a logic flow chart showing technical details of how variables and functions work or communicate	Yes/No/Not Applicable	High
1.5	The smart contract offers a transparent procedure about how the potential changes and upgrades will occur	Yes/No/Not Applicable	High
1.6	The smart contract is set to be deployed on a reputable Blockchain Network	Yes/No/Not Applicable	Critical
2. Smart Contract Development			
2.1	The smart contract code is open-source and has been peer-reviewed	Yes/No/Not Applicable	High
2.2	The smart contract has been audited before goes to production (e.g. deployed on a Blockchain)	Yes/No	Critical
2.3	The smart contract has been audited by an independent third-party security firm the last 6 months	Yes/No/Not Applicable	High
2.4	The Selected auditing firms are based on their expertise in blockchain technology, past audit reports, reputation in the industry, and contributions to security research	Yes/No/Not Applicable	High

2.5	Dependencies on external smart contracts or libraries have been minimized	Yes/No	High
26	The external smart contacts or libraries are regularly audited by a reputable security firm at least every 6 months	Yes/No	High
2.7	Functions and variables are named clearly and follow consistent naming conventions	Yes/No	High
2.8	Static analysis tools are used to identify common vulnerabilities (e.g., reentrancy, overflow/underflow etc.)	Yes/No	Critical
2.9	Known anti-patterns and bad practices (e.g., reliance on tx.origin, access control issues) are avoided.	Yes/No	Critical
2.10	Dynamic analysis has performed to uncover unexpected behavior	Yes/No	Critical
2.11	All identified issues from dynamic analysis have been addressed.	Yes/No	Critical
2.12	Fuzzing campaigns are conducted regularly, with custom fuzzers designed to generate inputs that target the specific logic of the smart contracts	Yes/No	Critical
2.13	Comprehensive unit and integration tests are conducted, covering all critical paths	Yes/No	Critical
2.14	Reputable frameworks related on secure by design developing of smart contracts are used (e.g. Smart Contract Security Verification Standard (SCSVS) to further check compliance in following industry's best practice	Yes/No	High
2.15	Unused dependencies and outdated code are removed from the codebase	Yes/No	High
2.16	The smart contract utilizes audited smart contracts	Yes/No/Not Applicable	High

	such as OpenZeppelin's Safemath to prevent overflow and underflow attacks		
2.17	Functions vulnerable to reentrancy attacks are protected with modifiers or audited smart contracts such as OpenZeppelin's Reentrancy Guard.	Yes/No/Not Applicable	High
2.18	If upgradable proxies are used, the implementation follows best practices to prevent storage collisions and ensure upgrade safety	Yes/No/Not Applicable	High
3. Access Control and Authorization			
3.1	The documentation explains briefly how access control works in the smart contract	Yes/No	High
3.2	The Company implements an Access Control Policy in both on-chain and off-chain management.	Yes/No	High
3.3	Ownership of the smart contract is clearly defined, with mechanisms to transfer ownership if necessary	Yes/No	High
3.4	Password strength Policy is being enforced in passwords related on Information Systems as well as protecting wallets	Yes/No	Critical
3.5	Access control mechanisms are correctly implemented to restrict actions to authorized users only.	Yes/No	High
3.6	Access Control mechanisms have been tested in various scenarios to adapt the new thread models	Yes/No	High
3.7	Multisignature wallets or similar schemes are used for protecting Access Control and restrict unauthorized acts	Yes/No/Not Applicable	Medium
3.8	Information Systems of the company as well as systems	Yes/No	High

	such as git are using only identifiable usernames		
4. Upgrades and Vulnerability Patches			
4.1	There is a clear and transparent governance process for proposing and implementing changes to the smart contract (e.g. a Vulnerability Patch Management Policy)	Yes/No/Not Applicable	Medium
4.2	Upgrades to new versions are handled securely	Yes/No	High
4.3	Before upgrading to the main product, upgrade has been tested in testnets to identify any unintentional anomaly after the upgrade	Yes/No	Critical
4.4	Upgrades are being audited and are trying to follow the logic that has been recorded on the corresponding documents	Yes/No	Medium
4.5	In case of critical upgrades which change mainly the architectural design of the smart contract, the chart is being redesigned to follow the new implementation	Yes/No/Not Applicable	Medium
4.6	Changes in the code are being recorded by a system such as git to identify what and from whom the change has occurred	Yes/No/Not Applicable	Medium
4.7	If upgradable proxies are used, the implementation follows best practices to prevent storage collisions and ensure upgrade safety	Yes/No/Not Applicable	Medium
4.8	The company is performing Penetration Tests every 6 months by reputable firms on its infrastructure to secure the development process of its smart contracts and its keys.	Yes/No	High
5. Security Awareness and Training			

5.1	Mandatory periodic security training sessions for all developers, focusing on the latest security practices, vulnerability trends and mitigation techniques. Is being executed annually.	Yes/No	High
5.2	The developers are being advised from platforms such as SWF to avoid common insecure development approaches	Yes/No	High
5.3	The developers are utilizing cyber intelligent tools to be informed about the latest cybersecurity news.	Yes/No	Low
6.Cryptography, Privacy and Randomness			
6.1	The smart contract briefly explains the cryptographic schemes that are being used	Yes/No/Not Applicable	High
6.2	The company implements policies and procedures related on cryptography for both on-chain and off-chain operations.	Yes/No	High
6.3	Cryptographic functions used in the smart contract are well-established and considered secure.	Yes/No/Not Applicable	Critical
6.4	The smart contract does not store any confidential, personal, or sensitive information on-chain.	Yes/No/Not Applicable	Critical
6.5	Data privacy measures (e.g., zero-knowledge proofs, off-chain storage with encryption) are implemented for private transactions	Yes/No/Not Applicable	Medium
6.6	The smart contract in order to achieve true randomness, it uses external verified and tested Oracles	Yes/No/Not Applicable	High
7.Compliance and Legal			
7.1	The smart contract complies with the relevant	Yes/No	High

	legal and regulatory requirements.		
7.2	Smart contract operations are compliant with data protection regulations (e.g., GDPR)	Yes/No	High
7.3	The smart contract has been certified from companies such as EEA Ethtrust	Yes/No/Not Applicable	Low
7.4	The documentation and the smart contract inform the users about the potential risks of its usage	Yes/No	High
8.1 Risk Assessment			
8.2	The company implements a Risk Management and Risk Treatment Policy	Yes/No	High
8.3	The developers of a smart contract are conducting regular Risk Assessments on smart contracts Functions	Yes/No	High
8.4	The developers analyze the risks associated of calling external smart contracts or oracles	Yes/No/Not Applicable	High
8.5	The Residual Risks are documented and communicated	Yes/No/Not Applicable	High
9.Monitoring			
9.1	The company implements an Event Logging Policy	Yes/No	High
9.2	The smart contract uses event calls to create logs	Yes/No	Critical
9.3	The company utilities specialized monitoring tools to identify any anomalies and common patterns.	Yes/No/Not Applicable	High
9.4	The company utilizes a SOC to monitor 24/7 the behavior of the smart contract	Yes/No/Not Applicable	Medium
10.Incident Response and Recovery			
10.1	There is a predefined incident response plan for security breaches, security bugs or other potential incidents that may occur	Yes/No/Not Applicable	High

10.2	The company has assigned responsibilities to certain persons in relation to the implementation of the incident response plans	Yes/No/Not Applicable	High
10.3	The smart contract provides mechanisms for funds recovery in case of theft or loss such as time locking	Yes/No/Not Applicable	High
10.4	Smart Contracts are being covered with insurance to protect against security risks	Yes/No/Not Applicable	High
11.Financial Security			
11.1	The smart contract offers mechanisms to prevent economic attacks, such as flash loan attacks.	Yes/No/Not Applicable	High
11.2	Security mechanisms in the Defi platform are well documented to provide further security and transparency		
11.3	Mechanisms are in place to manage liquidity risks, including automated liquidity monitoring.	Yes/No/Not Applicable	High
11.4	Mechanisms are in place to avoid smart contracts assumptions (e.g. user's balance, repayments) that can be used in attacks	Yes/No/Not Applicable	High
11.5	Collateralization ratios are regularly reviewed and adjusted based on market conditions.	Yes/No/Not Applicable	High
11.6	Price oracles are resistant to manipulation, using multiple sources and time-weighted averages	Yes/No/Not Applicable	Critical
11.7	Oracle failure or manipulation fallback procedures are documented and tested	Yes/No/Not Applicable	Critical

12.2. Examples of Testing and Using the Smart Contract Auditing Framework

In this chapter we will discuss some example usages which this Smart Contract Auditing framework can be utilized to support individuals, developers, companies and independent

Auditors. Each example is presented with a background scenario followed by a potential implementation which could be implemented in the auditing process.

Example 1: Designing a new startup company for developing a Defi application

Background: A new blockchain startup is looking to launch a new Defi application but is concerned about its compliance and building trust on its projects with new potential investors. It aims to create a more transparent model to show how their Defi application functions, how it uses data from oracles, how updates are managed and who can control the Defi application in terms of access control. The startup company has many well-trained developers and security experts.

Implementation: The startup utilizes this Smart Contract Auditing framework to ensure both technical security and find potential controls with things that can develop more transparency in its projects. As a result, the startup applies the Smart Contract Auditing framework, answers “Yes” or “No to the questions and identifies the risks associated with security, as well as policies, procedures and logical flow charts that they should design in order to be more trusted in the community.

Example 2: Auditing a Defi Project

Background: A reputable IT Auditing company is tasked with a project related to Smart Contract Auditing in a startup project. While the reputable IT Auditing company has hired senior security consultants, they want to standardize their Audit so as to be in agreement with the client and ensure the best result, identifying potential vulnerabilities and risks associated with this smart contract and recommend potential solutions to mitigate the risks.

Implementation: The reputable IT Auditing company utilizes this Smart Contract Auditing framework as the basis of the Audit and identify potential vulnerabilities and evaluate risk. Based on this framework, the reputable IT Audit company checks if the controls are satisfied and the document findings which the startup has to resolve to mitigate risks.

Example 3: Enhancing Security for an Established Defi Platform

Background: An established Defi platform plans to introduce a new set of complex financial products. Concerned about the potential security vulnerabilities that might arise with these additional in combination of changes in the logical flaw of the smart contract. The platform team has a lot of innovative and experienced developers but seeks a systematic approach to guarantee the highest standards of security as well as ensuring transparency in the users of the project due to the introduction of the new financial products.

Implementation: The Defi platform adopts the Smart Contract Auditing Framework to perform a comprehensive internal Audit and risk assessment. The framework guides the platform’s team through a detailed review process to identify the changes that they have to follow in their policies, procedures, data flow, patch management etc. and other areas that they need to improve, evaluating the existing smart contracts and processes. At the same time, the framework helps identifying specific risks associated with integrating new financial products, particularly focusing in designing logical flaw and smart contracts.

Chapter 13: Conclusions and Recommendations

This master thesis serves as a reference highlighting the critical need for enhanced security in Decentralized Finance and smart contracts and tried to combine the most critical parts from the regular IT Audits following industries' standards, to the Blockchain world. The main focus was to support companies in the blockchain industry to develop more secure decentralized applications and it included a way detailed information from different perspectives.

The research in keeping smart contracts secure and following and designing the best practices policies and procedures is a continuous effort to improve the security and it is a never-ending rat race. This master thesis included the most critical parts, they were analyzed contrary to normal IT audits to keep the best efforts. Furthermore, the Smart Contract Auditing framework which has been designed has tried to conclude as many control areas as possible, except the common vulnerability area, trying to support companies to mitigate risks and organize their infrastructure and access control better, understand the risks associated with activities in Blockchain.

The analysis of traditional IT auditing versus smart contract auditing has shown the unique challenges and opportunities inherent in securing blockchain-based applications showing how we approach security in an inherently decentralized environment. The exploration of DeFi topics, including ERC20 tokens, NFTs, flash loans, oracles and bridges, further demonstrated the critical need for enhancing security practices to safeguard against the dynamic threats in the blockchain world, where the modifications are not as easy as in the traditional IT systems.

As DeFi continues to evolve rapidly, the imperative for comprehensive smart contract auditing and the development of advanced security frameworks will be more and more mandatory to secure user's funds and trust in decentralized projects.

However, this study has some potential limitations. The Smart Contract Auditing framework designed in this master thesis has not been tested within a real-world company due to the nature of the blockchain industry. Consequently, we have not obtained data and feedback from practical implementation in a live environment. Despite this, we propose for this Smart Contract Auditing framework in the upcoming period to be tested extensively. This Smart Contract Auditing framework has been designed on further research conducted for this master thesis as well as the experience I have gained from working in IT Auditing in conventional companies and critical infrastructure over the past two years.

Despite these limitations, the Smart Contract Auditing framework makes substantial contributions to the field of Blockchain security as it offers a comprehensive approach to securing smart contracts and supporting organizations prevent, detect and patch potential threats more effectively and at the same time. Simultaneously, it promotes greater trust and transparency within the blockchain world. In addition to Smart Contract Auditing Framework, this master thesis explored the use of AI chatbots for applying static analysis in smart contracts. These chatbots are capable to identify vulnerabilities, recommend potential solutions and even autonomously patch these vulnerabilities.

Furthermore, the 18 new security awareness videos developed from the Hellenic speaking communities will significantly enhance knowledge in Greece and Cyprus about Blockchain and Smart Contract Development and Security. These videos aim to further promoting the Web 3.0 spirit in these respective countries, motivating more individuals to engage with and design solutions in Web 3.0 .

Based on the insights gained from this master thesis, it is recommended that companies operating in the blockchain space prioritize the establishment of transparent policies and procedures as well as flow charts, including charts that detail the logic of smart contracts to promote trust and transparency in the market. Moreover, companies and developers must perform comprehensive security audits to mitigate risks associated with common vulnerabilities. This includes adopting a proactive approach to security by integrating continuous auditing practices and engaging in regular security training for developers.

This master thesis has discussed many topics, however due to the rapidly evolving nature of Blockchain Technologies, the research is a never-ending approach. Implementing and adopting comprehensive security frameworks and even creating ISO27001 like certifications for Blockchain related companies and smart contracts, could further support the development of secure decentralized applications and build further trust in Blockchain world.

In conclusion, this master thesis not only provides a foundational understanding of the security challenges facing DeFi but also serves as a basis to call developers, auditors, and stakeholders within the ecosystem to prioritize security in every situation.

Furthermore, the market should invest more in the development of automated smart contracts tools which will leverage advanced technologies such as Artificial Intelligence and machine learning to support the developers and auditors in improving their tests.

For academia and future research for a wider role, there is a pressing need to explore even more advanced approaches in smart contract security, implementing more complex solutions that simultaneously support more transactions with higher throughput and lower gas fees. Additionally, investigating the impact of quantum computing on blockchain security presents a critical area of study, as smart contracts rely not only on their security, but also on the security of their blockchain infrastructure.

References

1) Solidity Docs: https://docs.soliditylang.org/
2) Remix IDE Docs: https://remix-ide.readthedocs.io/
3) Antonopoulos, Andreas M.; Wood, Gavin (2018). Mastering Ethereum: building smart contracts and DApps (First ed.). Sebastopol, CA: O'Reilly Media, Inc.
4) Token Economy: How the Web3 Reinvents the Internet (2020) Shermin Voshmgir
5) Eth.Research: https://ethresear.ch/latest
6) The Etheraut - A Web3 wargame played in Ethereum Virtual Machine https://ethernaut.openzeppelin.com/
7) Infura Ethereum Nodes: https://infura.io/
8) web3.js Documentation: https://web3js.readthedocs.io/en/
9) Metamask Guide: https://docs.metamask.io/guide/
10) Web3.py - Read The Docs: https://web3py.readthedocs.io/en/stable/
11) Remix IDE: https://remix.ethereum.org/
12) ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER - DR. GAVIN WOOD : http://gavwood.com/paper.pdf
13) Developing an Ethereum Based Blockchain Camera, Dimitris Vagiakakos (2022): https://github.com/sv1sjp/Blockchain_Camera/blob/main/Blockchain_Camera.pdf
14) Monitoring a Blockchain with a SIEM: https://medium.com/coinmonks/monitoring-a-blockchain-with-a-siem-6a0a0adf6ac5
15) Myhtril Docs: https://mythril-classic.readthedocs.io/
16) https://www.fatf-gafi.org/content/dam/fatf-gafi/brochures/opportunities-and-challenges-of-new-technologies-handout.pdf
17) ISO/IEC 27001:2022: https://www.iso.org/obp/ui/en/#iso:std:iso-iec:27001:ed-3:v1:en
18) The NIST Cybersecurity Framework 2.0 : https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.29.ipd.pdf
19) Hellenic Republic Cybersecurity Self Assessment Tool : https://mindigital.gr/wp-content/uploads/2022/03/cybersecurity-self-assessment.xlsm
20) https://medium.com/simform-engineering/implementing-multi-sig-wallets-in-smart-contracts-27c71a58d071
21) EEA EthTrust Security Levels Specification v-after-2 Editor's Draft https://entethalliance.github.io/eta-registry/security-levels-spec.html
22) https://app.forta.network/attack-detector
23) https://eips.ethereum.org/EIPS/eip-2535
24) https://cryptoconsortium.org/standards-2/
25) https://github.com/0xisk/awesome-solidity-gas-optimization
26) Auditing with Smart Contracts Andrea M. Rozario, Miklos A. Vasarhelyi
27) EIP-867 - Standardized Ethereum Recovery Proposals: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-867.md
28) https://www.immunebytes.com/blog/solidity-vulnerability-short-address-attack-in-evm/
29) https://ease.org/
30) https://tidal.finance/
31) https://nexusmutual.io/
32) https://www.alchemy.com/dapps
33) https://www.bitdegree.org/crypto/tutorials/ethereum-vs-ethereum-classic
34) https://www.investopedia.com/terra-5209502
35) On the detection of selfish mining and stalker attacks in blockchain networks: https://link.springer.com/article/10.1007/s12243-019-00746-2
36) PCI DSS Self-Assessment Questionnaire B-IP and Attestation of Compliance: https://listings.pcisecuritystandards.org/documents/PCI-DSS-v3_2-SAQ-B_IP-rev1_1.pdf
37) Proof Of Authority: https://academy.binance.com/en/articles/proof-of-authority-explained

38) Delegate Proof Of Stake: https://academy.binance.com/en/articles/delegated-proof-of-stake-explained
39) CryptoCurrency Security Standard (CCSS) - https://cryptoconsortium.org/standards-2/
40) Smart Contract Security Verification Standard - https://github.com/securing/SCSVS
41) ayer 1 vs. Layer 2: The Difference Between Blockchain Scaling Solutions https://www.investopedia.com/what-are-layer-1-and-layer-2-blockchain-scaling-solutions-7104877
42) Polygon Docs: https://docs.polygon.technology/pos/
43) Arbittrum Docs: https://docs.arbitrum.io/welcome/get-started
44) Optimism Docs: https://community.optimism.io/
45) https://medium.com/@j2abro/a-visual-guide-to-blockchain-bridge-security-e982fec671a7