NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Vulnerability Analysis of Ethereum Smart Contracts

Charalampos S. Maraziaris

Supervisor: **Konstantinos Chatzikokolakis,** Associate Professor

ATHENS

MARCH 2023

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ανάλυση Ευπαθειών στα Έξυπνα Συμβόλαια του Ethereum

Χαράλαμπος Σ. Μαραζιάρης

Επιβλέπων: **Κωνσταντίνος Χατζηκοκολάκης,** Αναπληρωτής Καθηγητής

ΑΘΗΝΑ

ΜΑΡΤΙΟΣ 2023

BSc THESIS

Vulnerability Analysis of Ethereum Smart Contracts

Charalampos S. Maraziaris
S.N.: 1115201800105

SUPERVISOR: **Konstantinos Chatzikokolakis,** Associate Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ανάλυση Ευπαθειών στα Έξυπνα Συμβόλαια του Ethereum

Χαράλαμπος Σ. Μαραζιάρης
Α.Μ.: 1115201800105

ΕΠΙΒΛΕΠΩΝ: Κωνσταντίνος Χατζηκοκολάκης, Αναπληρωτής Καθηγητής

ABSTRACT

Initially known as the underlying technology behind Bitcoin, the first and most widely recognized cryptocurrency, blockchain has evolved into a versatile tool with diverse applications beyond digital currencies. The distributed and decentralized nature of the blockchain allows for the creation of tamper-proof and transparent databases, enabling secure and reliable transactions without the need for intermediaries. This characteristic has propelled the adoption of blockchain in various fields, including finance, supply chain management, healthcare, and voting systems.

Ethereum, the second-largest blockchain platform after Bitcoin, has played a significant role in driving the growth and adoption of blockchain technology. Ethereum's programmable Smart Contract functionality enabled the creation of Decentralized Finance (DeFi) applications, which offer financial services such as lending, borrowing, and trading, without the need for banks. DeFi has emerged as one of the most promising use cases for blockchain technology, with the total value locked in DeFi applications exceeding tens of billions of dollars.

However, the security of smart contracts has been a significant challenge, as evidenced by the numerous high-profile hacks and exploits that have taken place in recent years. As the blockchain continues to grow and accumulate more funds, it becomes critical to evaluate and address its security vulnerabilities to ensure the trust and confidence of its users.

In this paper, we explore the security challenges facing blockchain-based systems, with a particular focus on Ethereum and its smart contract platform. We examine the different types of attacks that have been carried out against smart contracts and the underlying blockchain infrastructure, and we explore the various security mechanisms and best practices that can be employed to mitigate these risks. Our goal is to provide a comprehensive overview of the current state of security in the smart contract field and to offer insights into how we can develop more secure and robust coding practices in the future.

SUBJECT AREA: Blockchain, Cybersecurity, Vulnerability Analysis

KEYWORDS: Smart Contracts, Solidity, Ethereum, Vulnerabilities

ΠΕΡΙΛΗΨΗ

Η αλυσίδα κατανεμημένης εγγραφής (blockchain) έγινε αρχικά γνωστή ως η τεχνολογία πίσω από το Bitcoin, το πρώτο και πιο ευρέως διαδεδομένο κρυπτονόμισμα. Έκτοτε, έχει εξελιχθεί σε ένα ευέλικτο εργαλείο με ποικίλες εφαρμογές πέρα από τα ψηφιακά νομίσματα. Η κατανεμημένη και αποκεντρωμένη φύση της αλυσίδας κατανεμημένης εγγραφής χρησιμοποιείται για τη δημιουργία αδιάβλητων και διάφανων βάσεων δεδομένων, επιτρέποντας ασφαλείς και αξιόπιστες συναλλαγές χωρίς μεσάζοντες. Αυτό το χαρακτηριστικό έχει οδηγήσει στην υιοθέτηση της τεχνολογίας σε πεδία όπως τα οικονομικά, τα συστήματα υγείας, τα συστήματα διαχείρισης αλυσίδων εφοδιασμού και τα συστήματα ηλεκτρονικής ψηφοφορίας.

Το Ethereum, η δεύτερη μεγαλύτερη αλυσίδα κατανεμημένης εγγραφής μετά το Bitcoin, είχε πολύ σημαντική επίδραση στην ανάπτυξη και στην υιοθέτηση της τεχνολογίας. Η λειτουργικότητα των προγραμματίσιμων Έξυπνων Συμβολαίων που παρέχει το Ethereum οδήγησε στη δημιουργία των Εφαρμογών Κατανεμημένης Οικονομίας (ΕΚΟ), οι οποίες παρέχουν κλασσικές οικονομικές υπηρεσίες όπως δάνεια και ανταλλαγές, χωρίς να απαιτείται η παρέμβαση κάποιου τραπεζικού συστήματος. Οι ΕΚΟ αποτελούν μια από τις πιο υποσχόμενες εφαρμογές της τεχνολογίας, με τη συνολική αξία που έχει επενδυθεί σε αυτές να ανέρχεται σε πολλά δισεκατομμύρια δολάρια.

Ωστόσο, η ασφάλεια των έξυπνων συμβολαίων αποτελεί μια σημαντική πρόκληση, όπως καταδεικνύουν οι πολυάριθμες κυβερνοεπιθέσεις εναντίον τους τα τελευταία χρόνια. Όσο η τεχνολογία αλυσίδας κατανεμημένης εγγραφής αναπτύσσεται και συσσωρεύει περισσότερη αξία, τόσο πιο επιτακτική γίνεται η ανάγκη αποτίμησης και διόρθωσης των κενών ασφαλείας που προκύπτουν, ώστε να εξασφαλιστεί η εμπιστοσύνη και η υιοθέτησή της σε ευρεία κλίμακα.

Στην παρούσα εργασία εξερευνούμε τις προκλήσεις ασφάλειας που αντιμετωπίζουν συστήματα βασισμένα στην τεχνολογία αλυσίδας κατανεμημένης εγγραφής, επικεντρώνοντας τη μελέτη μας στο Ethereum και στα έξυπνα συμβόλαια που προσφέρει. Θα μελετήσουμε κάποιες κατηγορίες πραγματικών επιθέσεων που αφορούν τα έξυπνα συμβόλαια, και θα συζητήσουμε για τους διάφορους μηχανισμούς και τις πρακτικές προστασίας τους. Ο στόχος μας είναι να παρουσιάσουμε περιεκτικά την τωρινή κατάσταση στο πεδίο της ασφάλειας των έξυπνων συμβολαίων, καθώς και να παρέχουμε ιδέες που θα συνεισφέρουν στον ασφαλέστερο προγραμματισμό των εφαρμογών τους.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αλυσίδα Κατανεμημένης Εγγραφής, Κυβερνοασφάλεια, Ανάλυση Ευπαθειών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Έξυπνα Συμβόλαια, Solidity, Ethereum, Ευπάθειες

Στα παιδιά που έφυγαν νωρίς.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

The goal of the present thesis is to provide an analysis of common security vulnerabilities found in the development of Smart Contracts. The material discussed assumes that the reader has a background in Computer Science of undergraduate level and has no prior knowledge of concepts related to the field of Finance. We will provide an introductory description of the Ethereum blockchain, including the motivation behind the technology and its technical foundations in chapters 2 and 3. We will then describe the most popular and widely used services built on top of the Ethereum blockchain in chapter 4. Since many of these services revolve around finance, we will also introduce key financial concepts that play a major role in understanding the motivation and the means of exploitation behind the attacks we present. At the core of this work is chapter 5, where we examine four broad classes of smart contract vulnerabilities: reentrancy, missing access control, oracle manipulation, insufficient data validation. We will describe each vulnerability class and demonstrate real-world examples where a relevant vulnerability was exploited. In each examined attack we provide root cause analysis and mitigation recommendations. Through our analysis, we aim to extract common threat vectors that will enhance the security awareness in the broader field of smart contract development. The results of our analysis are summarized in chapter 6.

Since the source code performing the real-world attacks is not publicly available, we developed a public repository to assist our analysis. This repository provides a technical description of each attack, a live environment and sample exploit code to replicate the actual attacks and enhance the reader's understanding. The code is mainly written in the Solidity programming language and the repository can be located in the following URL:

https://github.com/cmaraziaris/smart-contract-vulnerability-analysis

2. BLOCKCHAIN FUNDAMENTALS

The **blockchain** is a distributed ledger technology commonly used as a data structure for storing information in a transparent and tamper-proof manner. It is a system that allows multiple parties to access the same copy of the ledger, without the need for intermediaries or a central authority. The ledger is maintained by a decentralized network of nodes, which validate and update it through a consensus mechanism.

2.1 Motivation

The concept of blockchain was first introduced in the paper “Bitcoin: A Peer-to-Peer Electronic Cash System” by Satoshi Nakamoto in 2008. [1] Blockchain technology was created as an alternative to the traditional digital finance that relies on central authorities, such as private and central banks, to verify and validate transactions. This centralized approach, while convenient and implemented almost since the inception of money, has several major drawbacks. First, it requires trust in a third party. The central financial institution has the power to censor transactions, either by blocking them or altering their details. This lack of transparency and accountability can lead to serious issues, especially in countries where the rule of law is not well established. Another disadvantage is that central authorities represent a single point of failure. If their systems fail or get compromised, the entire network of clients could be affected, with the impact ranging from a short-term denial of service to major loss of funds. Central authorities are also prime target for cyber attacks, since they hold large amounts of sensitive data required by laws and Know-Your-Customer (KYC) policies. Furthermore, traditional financial systems can be slow and inefficient, with transactions taking several days to be processed and verified. This can be frustrating for clients and allow fraudsters to exploit the system, given the large window of opportunity.

The blockchain technology was created to address these disadvantages by providing a decentralized, secure and efficient platform for digital transactions. Transactions are verified and validated through consensus mechanisms that do not rely on a central authority. This eliminates the need for trust in a third party, reduces the risk of censorship and single points of failure, and increases transparency and accountability.

2.2 Cryptographic Primitives

Cryptography forms the basis of mathematically proving the security and integrity of the blockchain technology. In this section we describe the basic cryptographic primitives we will encounter when we explain the components of a blockchain.

2.2.1 Hashing

Hashing is a fundamental concept in cryptography and is widely used for various security applications, such as password storage, digital signatures, and data integrity. It is a process of converting an input message of arbitrary length into a fixed-length string of characters, called a “hash value” or simply a “hash”. The key feature of hashing is that it is a one-way function, meaning that it is computationally infeasible to generate the original input message from its hash value.

On the implementation level, a hash function is used to generate a unique hash value for each input message. For a hash function to be considered cryptographically secure, it must have several specific properties. The first property is the avalanche effect, which means that a small change in the input should result in a drastic change in the output hash. This makes it difficult for an attacker to manipulate the input without being detected. Other important properties include the pre-image resistance, meaning that it should be computationally infeasible to generate an input that hashes to a specific value, and the collision resistance, which implies that it should be difficult for two different inputs to produce the same hash value. Finally, a secure hash function should have a high entropy, meaning that the output should be unpredictable and uniformly distributed. There are several widely used cryptographically secure hash functions, such as SHA-256. The selection of a suitable hash function depends on the specific requirements of the application, such as the desired level of security, the size of the hash value, and the processing time required to generate the hash.

2.2.2 Public Key Cryptography

Public key cryptography, also known as asymmetric cryptography, is a cryptographic system that uses two separate keys: a public key and a private key. The public key can be shared with anyone and is commonly used by other users to encrypt messages, while the private key is kept secret and used by the owner to decrypt the message.

Public key cryptography emerged as a solution to the scaling issue of symmetric cryptography. In symmetric cryptography, each pair of communicating users need to have their own shared secret key, with which they encrypt and decrypt messages. As a result, a network with n users would need n^2 keys in order to ensure the ability of each user to securely communicate with every other user. In a system that uses asymmetric cryptography where each user possess his private and public key, only $2n$ keys in total are required. For systems with millions of users, such as a blockchain, asymmetric cryptography is the preferred reliable and scalable solution.

The generation process of the key pair provides for the security of the public key cryptosystem. The pair is generated using mathematical algorithms that involve large prime numbers and the keys have the property that a message encrypted with one key can only be decrypted with the other key. It is also computationally infeasible to determine the private key from the public key, since the keys have a large enough key space to make brute-force attacks impractical, even with modern computing power.

2.2.3 Digital Signatures

Digital signatures play a critical role in ensuring the authenticity and integrity of electronic communications and transactions. By combining public key cryptography and hashing, they provide a secure and efficient means of verifying the identity of the sender and the integrity of the message, without relying on a trusted third party.

To create a digital signature, the sender first computes the hash of the message they want to send. They then encrypt the hash using their private key, creating the digital signature. The recipient of the message can then use the sender's public key to decrypt the signature and compare the decrypted signature with a hash computed from the received message. If the two hashes match, the recipient can be confident that the message came from the

sender and has not been altered during transit.

Examples of popular algorithms used for digital signatures include RSA, DSA, and ECDSA. RSA is one of the earliest and most widely used public key algorithms, DSA (Digital Signature Algorithm) is a U.S. Federal Government standard for digital signatures, while ECDSA (Elliptic Curve Digital Signature Algorithm) is a variant of DSA that uses elliptic curve cryptography for faster and more efficient key generation.

2.3 Blockchain Data Structure

At its core, blockchain is a data structure that resembles a linked list. It can be viewed as a continuous sequence of blocks, where each block contains a set of data. The blocks are linked together through cryptographic hashes, forming a chain. Each block in the blockchain contains a reference to the previous block's hash, creating an unbreakable link between the blocks.

This dependence on previous blocks is what makes the blockchain a secure and tamper-proof data structure. If someone tries to change the data in a block, the hash value for that block would change, and the hash value for all subsequent blocks would also change. This would cause a conflict in the blockchain and the change would be rejected, ensuring that the data remains intact.

In addition to its tamper-proof nature, the blockchain is also a distributed data structure. As a consequence, its data are stored on multiple nodes in the network, rather than being stored in a single location. This makes the blockchain highly resistant to downtime and failure, as the network can continue to function even if some nodes fail. The blockchain is also transparent, allowing all parties to view the chain, ensuring that there is no ambiguity about its contents.

The above properties ensure that all participants in the network can trust the integrity of the information stored in the blockchain, even if they do not trust each other.

2.4 Network

As we mentioned before, the blockchain is a decentralized data structure, meaning that it operates on a network of computers and there is no central authority that controls its operations. This concept is one of the main characteristics that sets blockchains apart from traditional data structures and databases.

The decentralized nature of the blockchain is achieved through the use of a **peer-to-peer network** of nodes. In a blockchain network, there is no central authority that manages the data and all the participants have equal responsibility for maintaining the network.

Each node in the network has a copy of the blockchain and participates in the consensus mechanism, which is a process to validate and add new transactions. When a transaction is added to the blockchain, it is broadcast to all the nodes in the network, who then independently validate the transaction and add it to their copy.

The **consensus mechanism** is an algorithmic process used to validate transactions. It is the key component of the underlying network that ensures the integrity and security of the blockchain.The most common consensus mechanisms used in blockchain networks are proof-of-work, proof-of-stake, and delegated proof-of-stake.

Proof-of-Work (PoW) is the original consensus mechanism used in the first blockchain, Bitcoin. In proof-of-work, nodes compete to solve a cryptographic puzzle, and the first node to solve the puzzle adds the next block to the blockchain. This process requires a significant amount of computational power, and it is designed to be difficult to achieve consensus without significant effort.

Proof-of-Stake (PoS) is a more energy-efficient consensus mechanism that involves nodes being selected to validate transactions and add blocks to the blockchain based on the amount of cryptocurrency they hold and are willing to "stake". Delegated proof-of-stake is a variation of proof-of-stake where nodes are selected to validate transactions based on a democratic voting process.

This decentralized structure means that there is no single point of failure in the network, as all the nodes work together to maintain the integrity of the blockchain. Transparency is also enhanced through this scheme, since it increases the difficulty for an attacker to compromise the network or manipulate the data stored on it. If one node fails, goes offline, or even gets compromised and becomes malicious, the network can continue to function as the other nodes take over its responsibilities.

3. ETHEREUM BLOCKCHAIN

The Ethereum [2] blockchain is an open-source blockchain platform that allows for the creation and execution of smart contracts and decentralized applications (dApps). It was first introduced in 2013, and it has since become one of the most widely used blockchain platforms in the world. The Ethereum network operates similarly to other blockchains, such as Bitcoin, with a decentralized network of nodes verifying and validating transactions, but it also has some unique features that set it apart. One key novelty is its use of a Turing-complete programming language, allowing developers to build complex, self-executing programs. This has led to an ecosystem of dApps built on top of the Ethereum network, including decentralized exchanges, prediction markets, and gaming platforms.

The main components of the Ethereum blockchain include:

- The Ethereum blockchain is made up of **blocks**, which are collections of verified transactions. Each block contains a unique identifier, the block hash, which links it to the previous block in the chain.
- **Nodes** are the computers that run the Ethereum software and validate transactions on the network. Nodes communicate with each other to validate transactions, reach consensus on the state of the blockchain, and maintain the integrity of the network.
- **Transactions** are the units of information that are stored on the Ethereum blockchain. Transactions can be any action that updates the state of the blockchain, such as sending or receiving cryptocurrency, executing smart contracts, or creating new accounts.
- **Ether** is the native cryptocurrency of Ethereum and is used to pay for the computational resources required to run smart contracts and to participate in consensus on the network.
- **Accounts** are entities on the Ethereum blockchain that hold balances of Ether and interact with smart contracts. There are two types of accounts: external accounts, controlled by private keys, and smart contract accounts, which are self-executing contracts that run on the Ethereum network.
- **Gas** is the unit of computational effort required to execute transactions and smart contracts on the Ethereum network. It is required to prevent spamming and ensure that the network is used efficiently.
- **Smart Contracts** are self-executing contracts that run on the Ethereum network and allow for complex logic to be executed on the blockchain. They provide a way for developers to build dApps that can interact with the Ethereum blockchain in a trustless manner.

In this chapter we will try to briefly describe each component.

3.1 Blockchain Structure

At a high level, the Ethereum blockchain is structured into blocks following the principles of the blockchain data structure introduced in the previous chapter. Each block contains a

series of transactions, which are processed and validated by the network's nodes. Each transaction models an interaction between accounts.

The main mechanism for holding, transferring and executing value on the Ethereum network is implemented through the use of **accounts**. In order to identify accounts in the Ethereum blockchain, each account is associated with a unique address, which is a 20-byte (160-bit) identifier. Addresses serve as a reference for the location of stored data, which can be Ether or smart contract code, within the Ethereum network.

Ethereum has two types of accounts: **External Owned Accounts (EOAs)** representing individuals and entities, and **Contract Accounts** representing autonomous code. Contract accounts are also called "Smart Contracts". Both types of accounts are associated with the monetary balance they currently hold in the form of Ether.

EOAs are the most common type of accounts on the Ethereum network and are often used by individuals, organizations, or other entities that want to participate in the Ethereum network. They are controlled by a private key, which is used to sign transactions sent to the network. These transactions allow an EOA to interact with the network and perform actions such as sending and receiving Ether or invoking a smart contract. The public key of the account is used to generate its Ethereum address. One important aspect of EOAs is their lack of state, meaning that they do not have any memory or storage capabilities.

In contrast, contract accounts are self-contained programs that exist on the Ethereum network and have their own unique address. Contract accounts are created when a developer deploys a smart contract to the network, and they are able to store data, interact with other contracts, and process transactions. Once deployed on the Ethereum blockchain, the code for a contract account can be executed by any node on the network and can interact with other contract accounts or EOAs. The address of a Contract Account is generated based on the code and the state of the contract when it was deployed. One of the key features of contract accounts is that they are autonomous and can operate independently, meaning that they can execute their programmed code and make changes to the blockchain state without requiring explicit instructions from their owners. These actions can be triggered by incoming transactions or messages. This type of account is typically used to implement various applications and services on the Ethereum network.

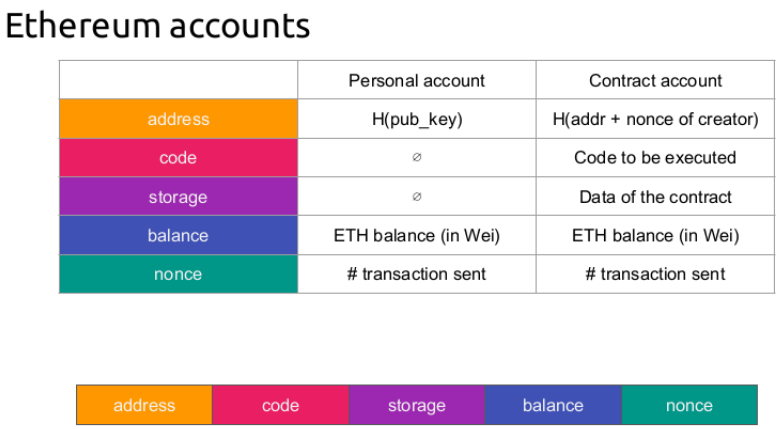

Figure 3.1: Structure of Ethereum accounts: EOAs and Contract Accounts

A **transaction** represents a transfer of value, data, or information between accounts. They are used to perform various operations such as transferring Ether, invoking a smart contract, or modifying the state of an existing contract.

Transactions contain several important pieces of information, including the sender and receiver addresses, the amount of Ether being transferred, and the data being passed along with the transfer. They also include a nonce, which is a unique identifier that is incremented for each transaction made by the same address.

Transactions in Ethereum are initiated by sending a signed message from an Ethereum account to another account. The signed message is then broadcast to the Ethereum network, where it is verified through its digital signature and processed by nodes on the network.

Once a transaction is included in the Ethereum blockchain, it cannot be reversed or altered. Therefore, transactions represent an unalterable and tamper-proof record of the transfer of value or information.

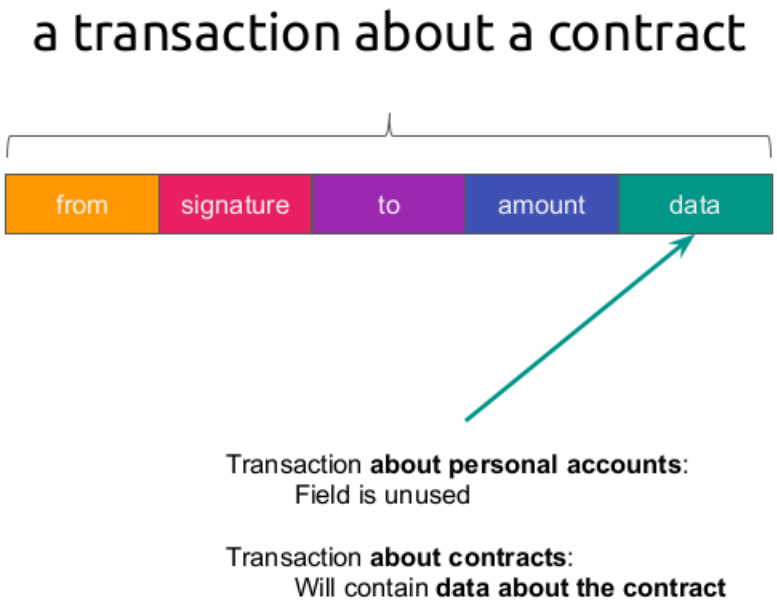

Figure 3.2: Structure of Ethereum transactions

Transactions are grouped into a **block**, which is then added to the existing blockchain infrastructure. A block in the Ethereum blockchain is a data structure that contains several fields of information, including a block header, a set of transactions, a set of receipts and the current state of the blockchain.

The block header includes information such as the block number, the block's hash, the previous block's hash, and the timestamp of the block's creation. The set of receipts contains one entry for each of the transactions included in the block. After a transaction is processed, a receipt is generated to provide evidence of the transaction's execution. The receipt contains information such as the status of the transaction, the amount of gas used during execution, and the contract address if a smart contract was involved. The set of transactions, set of receipts and the current state of the blockchain are represented using three different Merkle roots. For more information on the Merkle tree data structure and its variation used in the Ethereum network, the interested reader is advised to consult the slides of the blockchain course taught by Dimitris Karakostas. [3]

Each block in Ethereum contains a record of all the transactions and smart contracts that

have been executed by the time of the block's creation. By continuously adding new blocks to the chain, the Ethereum blockchain provides an immutable ledger of all the transactions that have taken place on the network.

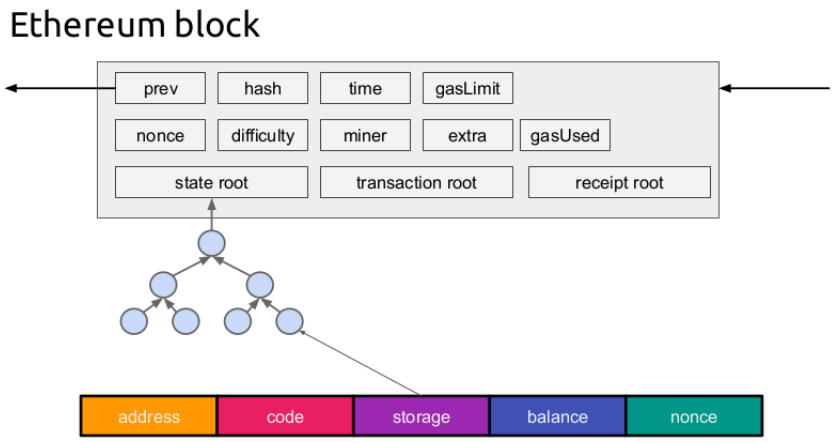

Figure 3.3: Structure of Ethereum blocks: State root encapsulates the blockchain state up to this block.

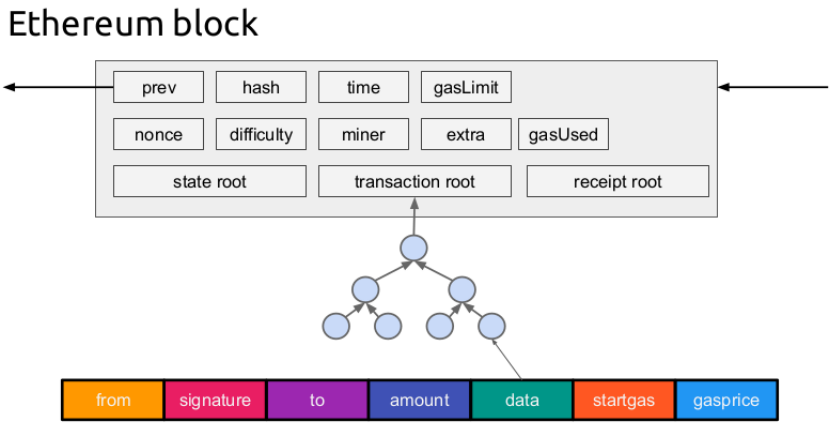

Figure 3.4: Structure of Ethereum blocks: Transaction root encapsulates the new transactions included in this block.

In the Ethereum network, new blocks are created and validated through a **Proof-of-Stake (PoS) consensus mechanism**. We will explain the steps of this algorithmic process.

Initially, a set of special network nodes called "validators" is selected. Validators are selected based on the amount of Ether they hold and are willing to "stake", which means to lock up, as collateral. This collateral acts as a guarantee that the validator will act honestly and follow the rules of the network. Therefore, validators are responsible for verifying transactions and adding them to the Ethereum blockchain in the form of blocks.

The specific algorithm used to select validators in Ethereum is called the "randomized slot protocol". Ether holders who want to become validators must stake a minimum amount of Ethereum as collateral. The more Ether a candidate stakes, the higher the likelihood that they will be selected as a validator, and the more rewards they can earn. The candidates collectively form a "validator pool". The pool of all validators is constantly changing, as new validators join and existing validators either leave or have their stake reduced due to

malicious behavior. At each block creation cycle, a set of validators are selected at random to create and validate a new block. The specific validators selected are determined by the combination of their stake and a random seed value, which is generated by the network. The random selection of validators helps to ensure that no single validator or group of validators has a disproportionate influence on the network.

From the set of validators, one validator is selected to create a new block by aggregating a set of pending transactions and broadcast it to the network. Other validators on the network then verify the block to ensure that it follows the rules and protocols of the Ethereum network. If a majority of validators on the network agree that the block is valid, it is added to the blockchain and becomes part of the permanent ledger of transactions. If a block is found to be invalid, it is discarded and the validator who created it may be penalized.

The block proposal process is prioritizing efficiency and security, ensuring that the Ethereum network can process a large number of transactions in a short amount of time. Additionally, by having multiple validators involved in the validation process, the network is able to ensure that no single validator or group of validators has control over the network, providing additional security and reliability to the Ethereum blockchain.

Validators are incentivized to act honestly, as they earn rewards for correctly validating transactions and creating new blocks, and these rewards come in the form of newly minted Ether. On the other hand, if a validator acts maliciously or violates the rules of the network, their staked funds can be penalized.

Ether (ETH) is the native cryptocurrency of the Ethereum blockchain. A cryptocurrency is a type of digital or virtual currency that uses cryptography to secure transactions and control the creation of new units. Contrary to fiat, government-issued currencies, cryptocurrencies are not necessarily backed by a central bank. ETH can only be generated, or “minted”, through the creation of a new block, as a reward for the validators that participated in its creation. Ether can be used as a store of value, much like traditional government-issued currencies and other cryptocurrencies, and also as an investment asset. It can be bought, sold, and traded on cryptocurrency exchanges, and it can be used to purchase goods and services from merchants who accept it as payment. Furthermore, it is used to pay for the computational resources required to execute transactions and smart contracts on the Ethereum network in the form of “gas”, a concept that we will introduce in the next section.

3.2 State Machine

Ethereum is often described as a state machine. A state machine is a mathematical model that describes the behavior of a system in response to a set of inputs, or “events”. By modeling Ethereum as a state machine, developers are able to build dApps on the Ethereum network that can perform complex computations and maintain a shared state of information, without relying on centralized servers or databases. [4]

In the context of Ethereum, the **state** refers to the current status of all information stored on the Ethereum network, such as: the current balance of all Ethereum accounts on the network, including both external and contract accounts, the code and data storage associated with each deployed smart contract, the number of transactions that have been sent from a particular address, the current status of any ongoing contract executions, event logs emitted by contracts during execution and the history of all transactions that have taken place on the network.

The state of the Ethereum blockchain is stored on every node in the network, and is updated with each block that is added to the blockchain. As new transactions and contract executions are processed, the state of the network is updated to reflect the current status of all accounts and contracts. All previous states are maintained in the blockchain, forming a permanent and tamper-proof ledger of all activity on the Ethereum network.

The **Ethereum Virtual Machine (EVM)** is responsible for maintaining the state of the Ethereum blockchain. It is a Turing-complete [5] virtual machine that executes the code of smart contracts and updates the state of the network as new transactions are processed. It acts as a virtual machine running in every node of the network, providing a secure and isolated environment for executing contract code, and it ensures that all rules and constraints defined in the smart contracts are followed. The memory, or storage, is modeled as a mapping of key-value pairs that represents the data stored in the contract.

Whenever a transaction is submitted to the Ethereum network, the EVM first checks the validity of the transaction and its associated data. If the transaction is valid, the EVM executes the code of the relevant smart contract, which compiles to EVM bytecode, updating the state of the network in the process. This could involve updating the balance of an Ethereum account, creating a new contract, or executing a function in an existing contract.

Once the EVM has executed the transaction and updated the state of the network, the changes are packaged into a new block along with other transactions and subsequently added to the blockchain. The EVM also maintains a copy of the current state of the network, which is updated as new blocks are added to the blockchain.

The EVM is designed to ensure that all transactions are executed in a secure, reliable, and tamper-proof manner. It also provides a consistent and deterministic environment for executing contract code.

Each operation performed by the EVM, such as writing data to storage or executing a contract, requires a certain amount of computational resources, such as CPU time, memory, and storage. To ensure that malicious or resource-intensive operations cannot overload the network, Ethereum uses a mechanism called **gas** to limit the amount of computational resources that can be consumed by any single transaction or smart contract execution. The same mechanism is used to reward the network nodes supplying the computational resources that validate and process transactions.

When a transaction is submitted to the network, it includes a **gas price**, which is the amount of Ether that the sender is willing to pay for each unit of gas. The gas price is used to calculate the total cost of the transaction in Ether, which is calculated by multiplying the gas price with the amount of gas required to execute the transaction.

The amount of gas required for a transaction or contract execution depends on several factors, such as the size of the transaction data, the complexity of the contract code, the number of operations required to execute the transaction and the network congestion. It is constrained by the **gas limit**, which specifies the maximum amount of gas that can be consumed by a single transaction or smart contract execution on the network. The gas limit acts as a cap on the amount of gas that can be consumed by all transactions and contract executions included in a block. It is set by the block validator for each block, and is included in the block header. If a single transaction or smart contract execution requires more gas than the gas limit, it will not be included in the block and the sender will receive an error indicating that the gas limit was exceeded.

The gas limit is a dynamic value that is adjusted by the Ethereum network over time, based

on the needs of the network. The aim is to set the gas limit high enough to support the computational needs of the network, while also ensuring that the limit is low enough to prevent resource exhaustion.

3.3 Smart Contracts

The **motivation** behind smart contracts is to provide a platform for creating decentralized, self-executing agreements that can automatically enforce the terms of a contract without the need for intermediaries. The idea is to create a trustless system where the terms of a contract are programmed into the contract code and can be executed automatically, reducing the risk of fraud, censorship, or interference from third parties. This aims to increase efficiency, reduce costs, and provide a more secure and transparent way of executing agreements compared to traditional contracts that rely on intermediaries and human intervention.

Smart contracts are **self-executing**, in the sense that they are executed automatically when specific conditions are met. As a real-world analogy, smart contracts can be compared to automated vending machines. Just as a vending machine has pre-determined rules (e.g. inserting a certain amount of money to get a certain item), smart contracts have predefined rules encoded in their code. When certain conditions are met (e.g. funds are received), the smart contract automatically executes the specified action (e.g. releasing the item). Just like vending machines, smart contracts eliminate the need for intermediaries (e.g. operators).

The **Ethereum Smart Contracts** are computer programs that run on the Ethereum blockchain. These programs are written in a high-level programming language, such as Solidity, and then compiled into machine-readable code that can be executed by the EVM. They contain the rules and conditions for executing a particular agreement or transaction between parties. The rules and conditions, encoded in their source code, are stored in the Ethereum blockchain, making them transparent, tamper-proof, and publicly accessible.

Smart contracts in Ethereum are created, or "**deployed**", using a transaction on the Ethereum blockchain. This transaction contains the smart contract's compiled bytecode, a low-level machine-readable format, which is then stored on the Ethereum state. Once the transaction is confirmed and added to a block, the smart contract becomes a part of the Ethereum network and can be executed by any node in the network. The deployment transaction also includes the code for the constructor, which is executed when the contract is first deployed, and sets the initial state for the smart contract.

When a smart contract is deployed to the Ethereum network, a new **contract account** is created to store its code and associated data. An EOA can then interact with the contract account by sending transactions to its address, which trigger its associated code to be executed. The execution of a smart contract can result in the creation of new contract accounts, the modification of its own state or the state of other contract accounts, and the generation of new transactions. It is important to note that a smart contract can only be triggered by an EOA or by other smart contracts.

Smart contracts are **executed** on the EVM, the sandboxed environment for executing code on the Ethereum blockchain that we described earlier. When a user wants to interact with the smart contract, they send a transaction to the Ethereum network that includes the desired action to be executed by the smart contract and the relevant function parameters as transaction data. The transaction is broadcast to the network and processed

by Ethereum nodes, which validate the transaction and add it to the blockchain.

Once the transaction is processed and included in a block, the EVM executes the smart contract by processing the bytecode associated with the contract. The EVM uses the current state of the Ethereum blockchain, including the current state of the smart contract, to execute the action specified in the transaction. If the execution of the smart contract requires additional computation or storage, it will consume gas, which is paid by the user who initiated the transaction.

The execution of the smart contract results in a new state of the Ethereum blockchain, which is then broadcast to the network and stored on all nodes. The new state of the smart contract and the Ethereum blockchain is then used as the starting point for any future transactions or interactions with the contract.

Smart contracts offer several **benefits** over traditional legal contracts, including:

- Autonomy: Smart contracts are self-executing, meaning that they can automatically enforce the terms of an agreement without the need for intermediaries. This reduces the risk of human error and ensures that contracts are executed as intended.
- Trust: Because smart contracts are stored on a decentralized blockchain, they are resistant to censorship, tampering, and fraud. This makes them a secure and trustworthy way of executing agreements.
- Transparency: Smart contracts are publicly accessible and transparent, allowing all parties to see the terms of the agreement and the progress of its execution. This promotes trust and accountability between parties.
- Efficiency: Smart contracts automate processes and reduce the need for intermediaries, which can save time and reduce costs.
- Programmability: Smart contracts can be programmed to perform complex calculations, access external data sources, and interact with other smart contracts, making them a versatile tool for a wide range of applications.
- Decentralization: Because smart contracts run on a decentralized blockchain, they can be accessed and executed from anywhere in the world, making them accessible to a global community of users and resistant to censorship.

These benefits make smart contracts a valuable tool for a wide range of applications, from financial transactions and supply chain management to digital identity and voting systems.

On the other hand, smart contracts have several **limitations**, including:

- Inflexibility: Once a smart contract is deployed, its code and terms cannot be changed. This can make it difficult to respond to changing circumstances or to correct mistakes.
- Complexity: Smart contracts are computer programs, and as such, they can be difficult to write, understand, and debug. This can create security risks and increase the likelihood of unintended consequences.
- Lack of legal recognition: In many jurisdictions, smart contracts are not yet recognized as legally binding agreements. This can create uncertainty and limit their adoption in certain industries.

- Limited data access: Smart contracts typically have limited access to external data sources and can only perform computations based on the information they receive. This can limit their ability to make complex decisions and interact with the real world.
- Scalability: As the number of users and transactions on the Ethereum network grows, the network can become congested and slow, leading to higher gas fees and longer confirmation times.
- Vulnerability: Smart contracts are only as secure as their code, and bugs or vulnerabilities in the code can lead to security issues and financial losses.

Overall, smart contracts can be used for a wide range of applications, including digital asset management, decentralized exchanges, crowdfunding and supply chain management. They provide a way for parties to interact and transact with each other in a trustless and efficient manner, without the need for intermediaries or centralized authority. It's important to note that once a smart contract is deployed on the Ethereum network, its code cannot be altered. This means that the rules and conditions defined in the contract are set in stone and cannot be changed. They can only be deleted under certain conditions. Therefore, it's crucial to thoroughly test and verify smart contract code before deploying it to the network to avoid any unintended consequences or errors.

3.4 Solidity

The most commonly used programming language for developing smart contracts on the Ethereum blockchain is **Solidity**, an open-source, contract-oriented programming language distributed by the Ethereum Foundation.

Solidity is statically typed and influenced by several other programming languages such as C++, Python, and JavaScript. Solidity is compiled into bytecode that can be executed on the EVM, which provides a secure and sandboxed environment for smart contracts to run. The EVM has its own set of opcodes (machine-level instructions) which can be used by Solidity to perform various operations on the Ethereum blockchain. The execution environment of Solidity is isolated from the underlying host system of the node and operates on the consensus mechanism provided by the Ethereum network, ensuring that smart contracts are executed in a secure and tamper-proof manner.

Some of the features of Solidity include inheritance, libraries, user-defined types, and the ability to interact with the Ethereum blockchain to perform operations such as sending Ether or reading data from the blockchain.

In Soldity, contracts are represented as classes in an object-oriented programming language. Each contract is composed of state variables, functions, and events. The state variables represent the current state of the contract, functions are used to interact with the contract and modify its state through transactions, and events are used to trigger actions in response to changes in the contract's state. Solidity also provides several built-in types and libraries to handle common tasks such as arrays, mappings, and string manipulation.

Solidity provides a specific syntax and programming paradigm that focuses on the creation and execution of smart contracts on the Ethereum blockchain. For this reason, it is classified as a **contract-oriented** programming language.

For a more in-depth presentation of its technical features, the interested reader shall refer to the official Solidity documentation. [6]

4. ETHEREUM SERVICES AND APPLICATIONS

The Ethereum blockchain offers a wide range of services and applications that utilize its rich features. In this chapter, we will explore some of the most popular and widely used Ethereum services and applications including tokens, decentralized exchanges and flash loans. These will lay the foundation upon which we build when discussing real-world exploits in later chapters.

4.1 Tokens

Tokens are digital assets that can represent a variety of things, including ownership of physical assets like real estate, or digital assets like in-game items or collectibles. They can also be used to represent a share in a company or organization, or a unit of value or currency. The motivation behind tokens in Ethereum is to allow for the creation and transfer of digital assets that can be traded, bought and sold, and used as a form of payment within the Ethereum network. They also provide a way for developers to build decentralized applications that have real-world value, such as digital collectibles, games, and decentralized exchanges. By using Ethereum as the underlying blockchain, developers have access to its infrastructure, which enables the creation of efficient and transparent token transactions.

Tokens in Ethereum are implemented as smart contracts. They are typically created through an initial coin offering (ICO), where investors exchange Ether or other cryptocurrencies for the newly created tokens. Once created, tokens can be traded and transferred on the Ethereum network, just like any other cryptocurrency. The token's properties, such as its supply and the rules for how it is traded, are encoded in the smart contract code and cannot be altered without consensus from the token's stakeholders. The token smart contract can be customized to suit the specific needs of the token, such as setting a fixed or a dynamic supply, implementing transfer restrictions or implementing different ways to interact with the token. The implementation of tokens as smart contracts on the Ethereum network has led to the creation of various token standards.

Token standards in Ethereum are created by the Ethereum community, including developers, users, and organizations. The standards are designed to provide a common set of rules and guidelines for the creation and deployment of tokens on the Ethereum blockchain. These standards ensure that tokens are interoperable and can be easily traded, transferred, and stored on the Ethereum network. The most widely adopted token standards are ERC-20 and ERC-721.

ERC-20 is a widely used token standard in Ethereum for representing fungible tokens, which are tokens that are interchangeable and have the same value, such as a currency. The ERC-20 standard defines a common set of functionalities that an Ethereum token must implement in order to be considered an ERC-20 token. This includes functions such as `transfer`, `approve`, and `transferFrom`, which allow the tokens to be transferred between Ethereum addresses. The standard also defines how the token's total supply, balance of an address, and other token-specific data are stored on the Ethereum blockchain.

ERC-721 is a token standard in Ethereum which defines a set of rules and functions that must be followed in order to create a non-fungible token (NFT). Unlike fungible tokens,

NFTs are unique and have a distinct identity, meaning they cannot be replaced by another token of equal value. ERC-721 allows developers to create NFTs with various properties and uses, such as digital collectibles, in-game items, or certificates of authenticity. The standard requires the implementation of specific functions for token ownership, transfer, and management, and it also enables NFTs to be listed and traded on decentralized exchanges and other marketplaces.

For more information on the ERC-20 and ERC-721 token standards, the interested reader shall refer to the official documentation. [7][8]

4.2 Decentralized Exchanges

Decentralized exchanges (DEXs) are trading platforms built on the Ethereum blockchain that enable users to trade cryptocurrencies and digital assets in a peer-to-peer manner, without the need for intermediaries or centralized third-party services, such as banks and brokers. Unlike traditional centralized exchanges, DEXs operate on a decentralized network, and all transactions are validated and executed on the Ethereum blockchain. This results in increased transparency and control for users, as their funds are stored in smart contracts on the blockchain, and can be accessed only by the user holding the private key to their wallet. In addition, DEXs offer greater freedom of choice, as they allow users to trade any ERC-20 or ERC-721 token that is supported by the Ethereum network, and are not subject to the same regulations and restrictions as centralized exchanges. This makes DEXs a popular choice for users seeking greater control and privacy over their digital assets. It should be noted that DEXs typically profit through transaction fees, which are taken as a percentage of each trade made on the platform and distributed to the contributors of the DEX.

DEXs are implemented as **smart contracts**, which are used to manage the exchange of assets, enforce trading rules, and store the order book. Since DEX trades are facilitated by deterministic smart contracts, they carry strong guarantees that they will execute in exactly the manner the user intended. In contrast to the opaque execution methods and potential for censorship present in traditional financial markets, DEXs offer strong execution guarantees and increased transparency into the underlying mechanics of trading.

In order to construct a trade, a trading mechanism, such as an order book and a trading engine, are required. For a trade to be actually implemented, a liquidity pool that will provide the relative assets is needed. An **order book** is a ledger that keeps track of all the buy and sell orders submitted by users on the DEX. It is maintained by the smart contract and is used to match buyers and sellers. The **trading engine** is the component of the DEX that matches buyers and sellers and executes trades. It uses the information in the order book and the assets in the liquidity pool to determine the best price for a trade and execute it automatically. **Liquidity pools** are collections of assets provided by liquidity providers that are used to facilitate trades. They provide the underlying mechanism for exchanging tokens on DEXs. A liquidity pool is essentially a smart contract on the Ethereum blockchain that holds a balance of two or more different tokens. The balance of the tokens in the pool is used to determine the price at which trades occur. The more liquidity, or assets, in the pool, the more stable the price of the tokens will be.

The operation of a liquidity pool can be understood as a decentralized, automated market maker. When a user wants to trade one token for another, they can submit a trade order to the liquidity pool. The smart contract then executes the trade based on the current prices

of the tokens in the pool. The amount of assets in the liquidity pool can change as users submit new orders, and liquidity providers can add or remove assets as needed.

Liquidity providers are individuals or entities that contribute assets to the liquidity pool. They play a critical role in the DEX, as they help to ensure that there is enough liquidity to facilitate trades and maintain stable prices. To incentivize users to provide liquidity to the pool, the platform typically rewards them with a portion of the trading fees generated by the pool.

These are the main concepts of a DEX in the Ethereum network. We will briefly discuss the implementation of these concepts in two of the most popular Ethereum DEXs: Uniswap and Curve.

Uniswap

Uniswap is a DEX built on the Ethereum blockchain that allows the exchange of ERC-20 tokens. It operates on an automated market maker (AMM) model, where instead of having an order book to match buyers and sellers, it uses a mathematical formula to determine the price of tokens. This allows for quick and easy trading without the need for intermediaries. When a user wants to exchange token A for token B, they send a transaction to Uniswap's smart contract providing the amount of token A to be exchanged and specifying a liquidity pool containing both tokens. The smart contract will place the sender's amount of token A in the liquidity pool, withdraw the amount of token B calculated by the AMM model and forward it to the sender. [9]

In the case of Uniswap, the AMM model uses the Constant Product Market Maker (CPMM) formula. In this model, the product of the reserves of all tokens in the pool is kept constant. The exchange rate between two tokens is determined by their ratio in the pool. When a user wants to trade between two tokens, the exchange rate is determined based on the current state of the liquidity pool and the trade is executed in a decentralized manner by automatically adjusting the reserves of the tokens in the pool, so that the constant product is maintained and the market is kept in balance. This design helps to reduce the impact of market volatility and provide a more stable and predictable trading experience. Additionally, the CPMM model used by Uniswap incentivizes liquidity providers by allowing them to earn trading fees proportional to the amount of liquidity they provide to the pool.

Curve

Curve is a decentralized exchange built on the Ethereum blockchain that provides a platform for trading **stablecoins**, which are cryptocurrencies pegged to the value of a stable asset such as the US dollar. It operates as a type of AMM platform, where users can trade cryptocurrencies in a trustless and decentralized manner. Curve aims to provide a platform that offers fast trading and stable pricing for its users, making it an attractive option for traders looking for an efficient way to trade stablecoins. [10]

One of the main novelties of Curve is its unique liquidity provision model, which uses a bonding curve to incentivize liquidity providers. This mechanism helps maintain a balanced supply and demand for each trading pair, leading to low slippage and low trading fees.

A **bonding curve** is a mathematical function that defines the relationship between the supply of a token and its price. This function takes into account the amount of tokens that have been issued, and the amount of capital that has been locked into the system, to calculate the price of each token. The most common type of bonding curve used in DeFi is the constant product bonding curve, which states that the product of the token supply

and price is constant.

In the context of the Curve DEX, the bonding curve is used to incentivize liquidity provision. The idea is that users who provide liquidity to the system by depositing a combination of assets are rewarded with a new token (typically labelled as *cToken*), which is created through the bonding curve. As more liquidity is added to the system, the supply of this new token increases, and the price of each token decreases. Conversely, as liquidity is withdrawn from the system, the supply of tokens decreases, and the price of each token increases.

To implement the bonding curve in the Curve DEX, the first step is to define the constant product formula, which takes the form of: $S \cdot P = k$, where S is the token supply, P is the token price, and k is a constant. Next, the Curve smart contract is deployed on the Ethereum blockchain, which implements the bonding curve formula and manages the token creation, transfer, and redemption process.

When a user wants to provide liquidity to the system, they deposit a combination of assets into the smart contract, and in return, receive an equivalent amount of the new token. The amount of tokens received is proportional to the amount of assets deposited, and is determined by the constant product formula. The tokens can then be traded on the Curve DEX, or held as a form of investment. Additionally, Curve implements a governance mechanism that allows token holders to propose and vote on changes to the exchange, giving the community more control over its development and future direction.

Overall, the combination of its bonding curve model, low fees, and community governance make Curve a unique and attractive option for those looking to trade stablecoin-pegged cryptocurrencies in a decentralized setting.

For more technical details regarding the implementation of Uniswap and Curve, the interested reader shall refer to the official documentation of these protocols. [9][10]

4.3 Flash Loans

Flash loans are a type of short-term, uncollateralized loan in the DeFi space. They are called “flash” loans because they can be taken and repaid in a matter of seconds, as opposed to the days or weeks required for a traditional loan.

In the context of finance, **collateral** refers to assets that are used to secure a loan. The loan is essentially backed by the value of the collateral, and if the borrower is unable to repay the loan, the lender can seize the collateral as compensation. The purpose of collateral is to reduce the risk of default for the lender and provide them with some level of assurance that they will be able to recover an asset of equivalent value if the loan is not repaid. This is particularly important in decentralized finance, where loans are typically made without the involvement of traditional financial intermediaries, and there is no central authority to enforce repayments.

In order to better understand flash loans we will compare them to traditional loans. Traditional loans typically involve a borrower receiving a loan from a lender and paying it back over a certain period of time, plus interest. On the contrary, flash loans are short-term loans that are taken out and repaid usually within a matter of seconds. The key difference between flash loans and traditional loans is that flash loans do not require collateral. Instead, flash loans are secured by the underlying blockchain properties, which enforce the repayment of the loan in the same transaction. This means that the borrower must

repay the loan in full, including any interest fees, before the transaction can be committed to the blockchain. Traditional loans are subject to a wide range of regulations and restrictions, such as credit checks and interest rate caps. Flash loans, on the other hand, are completely unregulated, meaning that they offer a high degree of flexibility and freedom to borrowers.

Flash loans are implemented using smart contracts on the Ethereum blockchain. The basic idea is to allow users to temporarily borrow funds without having to provide any collateral. This is made possible by the fact that all transactions on the Ethereum network are **atomic** and cannot be rolled back once executed. Therefore, a flash loan is initiated by a user sending a transaction to a smart contract that implements a flash loan provider. The flash loan provider will then automatically issue the loan based on the terms specified in the smart contract. The loan amount is deducted from the flash loan provider's liquidity pool, and the user can immediately use the funds. Once the user has finished using the funds, they must repay the loan by sending the original loan amount back to the flash loan provider's smart contract, which will then be added back to the liquidity pool. The request and the repayment of the loan must all happen within the same transaction. Otherwise, the loan provider is able to revert the transaction, which will automatically cancel the loan.

One of the most popular flash loan providers in Ethereum is **Aave**. Aave is a decentralized, non-custodial money market protocol that enables users to lend and borrow assets without intermediaries. It was the first platform to introduce flash loans on Ethereum. The core functionality of Aave is based on smart contracts that define the terms of the loans, such as the interest rate, the loan duration, and the amount that can be borrowed. They also enforce the rules of the protocol and automatically settle loans when they are due. Users who wish to lend their assets on Aave deposit them into a smart contract, which acts as a liquidity pool. Borrowers can then request a loan from this pool, and the protocol matches them with available funds. Incentives for participating in Aave can be found in various parts of the platform's design. Firstly, users are able to earn interest on their deposited assets by providing liquidity to the platform's various lending pools. In addition, the same users are able to borrow assets with a lower interest rate. The interest rate for borrowing is market-driven, meaning it is determined by the supply and demand of the available assets. Another incentive for participating in Aave is its unique governance system, which allows token holders to vote on platform upgrades and changes. This allows users to participate in the direction and development of the platform, which can lead to increased confidence and trust in the platform's long-term viability. For more information on the technical specification and implementation of the Aave protocol, the interested reader shall refer to the official documentation. [11]

To summarize, one of the main advantages of flash loans is their speed and ease of use. Flash loans can be taken out and repaid in a matter of seconds, allowing users to take advantage of market inefficiencies or execute financial strategies with a high degree of flexibility and speed. Additionally, because flash loans do not require collateral, they are accessible to a wider range of users, including those who do not have assets to pledge as collateral. Another advantage of flash loans is their security. The fact that flash loans must be repaid in a single transaction, and within a specific time frame, helps to ensure that users do not over-extend themselves and become unable to repay the loan. The borrowing and lending process is automated, and when everything works out, both the lender and borrower benefit from the loan. If anything goes wrong, the transaction is canceled, as if it never happened, and there's no profit for either one of the parties.

4.4 Mixers

Crypto mixers, also known as **tumblers** or **shufflers**, are privacy-focused tools used in the cryptocurrency space to obscure the source and destination address of a transaction. They work by taking in cryptocurrency from a user, mixing it with other transactions and then sending the mixed funds to a specified recipient address. This is done in an effort to break the connection between the original source of the funds and the recipient, making it more difficult for others to track the flow of funds. One of the most popular crypto mixer in the Ethereum network is *Tornado.Cash*. [12]

Tornado.cash is a decentralized platform that provides privacy to Ethereum transactions by allowing users to deposit their funds into a smart contract, and then withdraw these funds through one or many different addresses. The platform utilizes cryptographic zero-knowledge proofs to enable the exchange of tokens without revealing the identities of the parties involved or the amounts being transferred. In a zero-knowledge proof, one party can prove to another party that they know a certain piece of information, without revealing the information itself. Withdrawing funds from Tornado.cash requires creating a “withdrawal proof”. The **withdrawal proof** is a piece of information that the user generates, which proves that they own the funds they want to withdraw. To generate a withdrawal proof, the user must have access to their private key and the original deposit information, including the deposit transaction ID, the deposit index, and the deposit note. These pieces of information are used to create a unique proof that proves the ownership of the funds being withdrawn. This proof is then submitted to the Tornado.cash smart contract, which verifies its validity and releases the funds (or a part of them) to the specified withdrawal address.

In summary, crypto mixers are often used to increase privacy and security for users who are concerned about the potential for their cryptocurrency transactions to be tracked or traced. However, it is important to note that using a mixer does not guarantee complete anonymity, as other factors such as IP address and transaction metadata can still be used to identify individuals. Additionally, the use of mixers is often viewed as controversial, as they can also be used for illicit purposes such as money laundering or the financing of criminal activities, as we’ll see in the exploits we describe later.

5. SMART CONTRACT SECURITY

Smart contracts, self-executing computer programs, are the cornerstone of blockchain technology and the foundation of decentralized applications that aim to revolutionize various industries. However, with the increase in usage and adoption of smart contracts, security issues have become a significant concern. Numerous incidents of smart contract vulnerabilities have resulted in the loss of millions of dollars, highlighting the need for robust security measures. As the number and complexity of smart contracts continue to grow, it is essential to assess the current state of smart contract security, identify existing vulnerabilities, and explore potential solutions to improve security. This chapter aims to provide an overview of the current state of smart contract security, analyze recent attacks and vulnerabilities, and discuss potential mitigation techniques and best coding practices.

In the sections that follow, we examine four broad classes of vulnerabilities commonly found in smart contract development: reentrancy, missing access control, oracle manipulation and insufficient data validation. At the beginning of each section, we define what constitutes a vulnerability of this class and give a simple code example, where required, showcasing the problem. We also provide general mitigation techniques, often in the form of code snippets. At the core of each section, we present a number of real-world hacks exploiting a vulnerability of the examined class. The presentation includes a root cause analysis of the vulnerability, with pointers to the available source code, and mitigation recommendations applicable on each specific smart contract. The time frame of the attacks ranges between 2020 and 2022. We chose to examine attacks with financial impact ranging from a few thousand US dollars to multi-million losses.

5.1 Reentrancy

One technical feature of the Ethereum smart contracts is the ability to call functions and utilise code from other external contracts. When a call to another contract is made, the control flow is naturally passed down to that contract, so it can execute its code.

A reentrancy vulnerability can occur when a contract A makes an external call to another contract B, and the called contract B invokes the original contract A again before the first call is finished. This may cause the different invocations of contract A to interact in undesirable ways, especially if the calling contract makes the implicit assumption that external calls will not redirect the control flow back to itself.

The vulnerability can be exploited if the state of the original contract changes before the function is completed. If an attacker can call back into the function before it completes, they could possibly take advantage and manipulate the contract's new state. For example, they might be able to withdraw funds multiple times before the balance is updated, resulting in the theft of additional funds.

The following example demonstrates a smart contract vulnerable to reentrancy and the relevant malicious contract exploiting the vulnerability to drain its funds.

```
1  contract InsecureBank {
2      mapping(address => uint256) balances;
3
4      function withdraw(uint256 amount) public {
5          // This check always succeeds because the balance is never
6          // updated before the nested calls.
```

```
7           if (balances[msg.sender] >= amount) {
8               msg.sender.call{value: amount}("");
9               balances[msg.sender] -= amount;
10          }
11      }
12
13      function deposit() public payable {
14          balances[msg.sender] += msg.value;
15      }
16  }
17
18  contract Attacker {
19      InsecureBank bank;
20
21      constructor(address _bankAddress) public {
22          bank = InsecureBank(_bankAddress);
23      }
24
25      function attack() public payable {
26          // deposit some initial funds
27          bank.deposit.value(1 ether)();
28
29          // call the withdraw function to begin the reentrancy attack
30          bank.withdraw(1 ether);
31      }
32
33      function () external payable {
34          // this fallback function is called when the withdraw function
35          // in the bank contract sends ether to this contract
36          if (bank.balance >= 1 ether) {
37              bank.withdraw(1 ether);
38          }
39      }
40  }
```

In this example, the `withdraw` function allows a user to withdraw a certain amount of Ether from their balance in the `InsecureBank` contract. However, the `call` function is used to transfer the funds to an external address, which is the caller's address stored in `msg.sender`. In general, `call` is used to invoke a function implemented on a specific blockchain address, if the address belongs to a Contract account, or to send Ether if the address belongs to an EOA. If the function specified by the `call` argument is not implemented, then the default, "fallback", function of the receiver contract is executed.

The implementation above allows an attacker to re-enter the vulnerable contract before the balances mapping is updated, enabling them to withdraw more funds than they actually have. This is because the `call` function will execute the fallback function of the external contract before returning to the original contract, which in turn will execute a fresh invocation of the `withdraw` function.

We are going to give a detailed explanation of the `attack` function that performs the exploit. In the beginning, the attacker deposits 1 Ether to `InsecureBank`. Then, the attacker calls the `withdraw` function in the `InsecureBank` contract to ask for a one-Ether withdrawal. `InsecureBank` uses `call` to send the Ether and does not attach any function signature. The attacker's fallback function responds to this call by calling `InsecureBank`'s `withdraw` function again. The second call is regarded as a "nested" call inside the previous withdrawal call because the previous one has not finished yet. Normally, the caller will receive one Ether, and the caller's balance will be deducted as the execution of the instruction

`balances[msg.sender] -= amount;`. However, in a reentrancy situation, this instruction will never be executed because the `call` function above this line will lead to recursively calling the `withdraw` function until the total balance of `InsecureBank` is less than one Ether, eventually draining the contract of its funds.

Two **mitigating** techniques are commonly recommended, the Checks-Effects-Interactions pattern and reentrancy guards/locks. These two techniques should both be in place to ensure a defence-in-depth approach to prevent reentrancy vulnerabilities.

The **Checks-Effects-Interactions (CEI) pattern** is based on three principles:

1. Checks: Perform all necessary checks and validation of input parameters before modifying the state of the contract or interacting with other contracts. This step is important to ensure that the contract state is consistent and to prevent unexpected behavior.
2. Effects: After all checks have been completed, update the contract state. This step should be deterministic and not involve any external calls, to prevent race conditions or unexpected behavior.
3. Interactions: Finally, perform any necessary external interactions. This includes sending ether or tokens to other contracts, calling external functions, or emitting events. This step should be performed after the state has been updated, to prevent reentrancy attacks.

A **reentrancy guard or mutex lock** ensures that a function is executed atomically and cannot be reentered by an attacker. The logic is that a boolean “lock” variable is placed around the function call that is vulnerable to reentrancy. Every time the function is called, the guard is checked. The body of the function is executed only if the guard is in the “unlocked” state. Otherwise, the call is rejected. The initial state of the guard is “unlocked”, and when the function is first called, the guard becomes “locked”, disallowing further calls. When the function is about to exit, the guard becomes “unlocked” again. It is interesting to note that the guard logic can be extended from a specific function to sets of functions, including the whole contract.

The following code shows the two mitigation techniques discussed being applied to the vulnerable Bank contract:

```
1  contract SecureBank {
2      mapping(address => uint256) balances;
3      bool entered = false;
4
5      function withdraw(uint256 amount) public {
6          // Guard: If `entered` is true, the following line rejects the call.
7          require(entered == false, "Reentrancy attempt!");
8          entered = true;
9          if (balances[msg.sender] >= amount) {    // Checks
10             balances[msg.sender] -= amount;      // Effects
11             msg.sender.call{value: amount}("");  // Interactions
12         }
13         entered = false;
14     }
15
16     function deposit() public payable {
17         balances[msg.sender] += msg.value;
18     }
19 }
```

Next, we are going to discuss three real-world exploits due to reentrancy vulnerabilities.

5.1.1 Akropolis

Akropolis is a DeFi protocol that provides a range of financial services, such as lending, borrowing, and investment management. One of the key features of the Akropolis protocol is the ability to create and manage multiple yield-generating strategies within a single smart contract. This allows users to invest their funds in a diversified portfolio of assets and strategies, which can help to reduce risk and increase returns. [13]

The Akropolis protocol was hacked on the **12th of November 2020**, for a total loss of **2M USD**. The attacker managed to execute a $50,000 exploit 40 times, in a flash-loan assisted reentrancy exploit. The stolen funds were drained from Akropolis' Curve liquidity pools, which supply the project with liquidity, and were never returned. Before the attack, Akropolis underwent **two security audits**. [14][15]

The **root causes** of this attack were that the protocol did not validate the supported tokens and it did not enforce reentrancy protection on the deposit logic. The first factor allowed the attacker to register a deposit of a custom ERC20 token. By calling `transferFrom` on this token, the vulnerable contract passed the control flow to the malicious contract, which exploited the second factor to perform a second deposit, this time submitting an amount of a legitimate ERC20 token. However, the vulnerable contract did not implement the CEI pattern. Instead, the deposited amount is calculated based on the pool's balances before and after the deposit call, and the depositor receives rewards based on this calculation. Therefore, the sender of the second deposit (the malicious ERC20 token) receives rewards equal to its legitimate deposit amount. The interesting point is that the sender of the first deposit (the adversary) also receives rewards equal to the legitimate deposit, since the difference in balance before and after the first deposit call is equal to the legitimate deposit performed by the reentrant call. In summary, the attacker actually deposits a fixed amount in the vulnerable contract but obtains rewards equal to double of this amount.

```
1   function deposit(address _protocol, address[] memory _tokens, uint256[]
        memory _dnAmounts)
2   public returns(uint256)
3   {
4       PoolToken poolToken = PoolToken(protocols[_protocol].poolToken);
5       // [...]
6       uint256 nBalanceBefore = distributeYieldInternal(_protocol);
7       // VULN: Reentrancy happens here
8       depositToProtocol(_protocol, _tokens, _dnAmounts);
9       uint256 nBalanceAfter = updateProtocolBalance(_protocol);
10      uint256 nDeposit = nBalanceAfter.sub(nBalanceBefore);
11      // [...]
12      poolToken.mint(_msgSender(), nDeposit);
13      return nDeposit;
14  }
15
16  function depositToProtocol(address _protocol, address[] memory _tokens,
        uint256[] memory _dnAmounts) internal {
17      for (uint256 i=0; i < _tokens.length; i++) {
18          address tkn = _tokens[i];
19          // VULN: Reentrancy happens in the `transferFrom`
20          // which calls `deposit` again
21          IERC20(tkn).transferFrom(_msgSender(), _protocol, _dnAmounts[i]);
22          IDefiProtocol(_protocol).handleDeposit(tkn, _dnAmounts[i]);
23          // [...]
```

```
24          }
25      }
```

The attack is effectively **mitigated** by using a reentrancy guard in the `deposit` function, as showcased in the introductory example.

5.1.2 CREAM Finance

CREAM Finance is a decentralized lending and borrowing protocol built on the Ethereum blockchain. It was launched in August 2020 and enables users to lend and borrow a wide range of cryptocurrency assets. CREAM stands for “Crypto Rules Everything Around Me”, a play on the acronym “C.R.E.A.M.” originating from a Wu-Tang Clan song. From a protocol perspective, CREAM finance is similar to other DeFi lending protocols such as Aave. A unique feature of CREAM Finance is its customizability. The protocol is built on Compound Finance’s codebase , but it has been modified to provide more flexibility and customization options. For example, it allows users to set their own interest rates and collateralization ratios, providing more control over the borrowing and lending process. Additionally, CREAM Finance has implemented a “hybrid” governance model that allows both token holders and community members to participate in the decision-making process. [16]

The CREAM Finance protocol was exploited on the **31st of August 2021** for a total loss of **18.8M USD** user funds, over the course of 17 transactions. The stolen funds were not returned by the attacker. However, CREAM provided liquidity equivalent to the previous balances of the affected users and used a part of the protocol fees to gradually replace the stolen funds. It is worth noting that CREAM Finance was last audited on the 28th of January 2021, but the vulnerability was introduced on the 10th of February 2021, originating from a governance proposal. [17][18]

The **root cause** of this attack stems from a reentrancy vulnerability with the help of the ERC-777 token pre/post-transfer hooks. The ERC-777 token standard allows the execution of pre-determined functions (called hooks) before or after funds are received from an ERC-777 compliant contract. The CREAM Finance smart contracts were oblivious to this functionality and assumed that state-changing code would not be executed through an external `transfer` call. This behavior allowed an adversary to register malicious hooks that enabled him to nest two `borrow` calls. The second `borrow` call, allowed the attacker to borrow ETH against the same collateral used for the token of the first `borrow` call, since the effects of the first `borrow` were not updated in the contract. Therefore, by depositing collateral matching his initial borrow amount, he was able to obtain double the requested borrow amount and walk away with the profit.

```
1  /* `crETH` token */
2      function borrow(uint borrowAmount) external returns (uint) {
3          return borrowInternal(borrowAmount);
4      }
5
6      function borrowInternal(uint borrowAmount) internal nonReentrant returns (
           uint) {
7          uint error = accrueInterest();
8          if (error != uint(Error.NO_ERROR)) {
9              // accrueInterest emits logs on errors, but we still want to log
                   the fact that an attempted borrow failed
10             return fail(Error(error), FailureInfo.
                   BORROW_ACCRUE_INTEREST_FAILED);
```

```
11          }
12          // borrowFresh emits borrow-specific logs on errors, so we don't need
                to
13          return borrowFresh(msg.sender, borrowAmount);
14      }
15
16      function borrowFresh(address payable borrower, uint borrowAmount) internal
            returns (uint) {
17          // [...]
18          ////////////////////////
19          // EFFECTS & INTERACTIONS
20          // (No safe failures beyond this point)
21
22          /*
23           * We invoke doTransferOut for the borrower and the borrowAmount.
24           *  Note: The cToken must handle variations between ERC-20 and ETH
                 underlying.
25           *  On success, the cToken borrowAmount less of cash.
26           *  doTransferOut reverts if anything goes wrong, since we can't be
                 sure if side effects occurred.
27           */
28          doTransferOut(borrower, borrowAmount);  // EXPLOIT HAPPENS HERE
29
30          /* We write the previously calculated values into storage */
31          accountBorrows[borrower].principal = vars.accountBorrowsNew;
32          accountBorrows[borrower].interestIndex = borrowIndex;
33          totalBorrows = vars.totalBorrowsNew;
34
35          /* We emit a Borrow event */
36          emit Borrow(borrower, borrowAmount, vars.accountBorrowsNew, vars.
                totalBorrowsNew);
37
38          /* We call the defense hook */
39          comptroller.borrowVerify(address(this), borrower, borrowAmount);
40
41          return uint(Error.NO_ERROR);
42      }
43
44      function doTransferOut(address payable to, uint amount) internal {
45          /* Send the Ether, with minimal gas and revert on failure */
46          to.transfer(amount);
47      }
```

The first factor that enabled the attack, is that the contract logic did not account for the ERC-777 token transfer hooks. The protocol keeps a registry of allowed tokens that can be traded and most of them are ERC-20 compliant. Therefore, the implemented logic in the borrow functions is built around the whitelisted ERC-20 tokens, which do not perform external calls and thus are not supposed to reenter the calling function. However, external calls are permitted through ERC-777 transfer hooks, and thus the attacker used a whitelisted ERC-777 token introduced through a government proposal, to exploit the vulnerability.

The second interesting factor that enabled the attack is that **a reentrancy guard was implemented** and used in every smart contract that is part of the protocol. The CREAM Finance protocol manages a variety of tokens, and each of them is handled by a different smart contract. However, each reentrancy guard is **local** to the smart contract in use. The vulnerability can be attributed to a “protocol-level” reentrancy: each `borrow` function used in the exploit was protected from reentrancy (that is, not being able to call itself), however

there was no rescriction disallowing it to perform an external call to another token's `borrow` function, that was also part of the protocol. The adversary exploited this subtle omission, combined with the contract's failure to comply with the CEI pattern.

This attack is **prevented** by securing the `borrow` functions of each token contract with a **global, protocol-level** reentrancy guard. Additionally, following a more robust approach on the CEI pattern, the bookkeeping on the borrowed assets for each user should be performed **before** the transfer takes place, and should be restored if the transfer fails.

As will be evident in the analysis of the next attack, the protocol should actually protect every publicly exposed function with a global reentrancy guard, not just the `borrow` function.

5.1.3 Rari Capital, FEI Protocol

Rari Capital is a DeFi protocol that allows users to earn interest on their cryptocurrency holdings through a range of liquidity pools. The protocol provides a user-friendly interface that aggregates liquidity from various DEXs like Uniswap and Curve.

Rari Fuse, a product of Rari Capital, is a decentralized platform that allows users to create custom pools of assets with various strategies and risk profiles. Fuse provides the necessary infrastructure for users to create and manage their own pools, which can be tailored to suit their investment goals and preferences. These pools can be used for a variety of purposes, such as yield farming, liquidity provision, or asset management.

The Fei Protocol is a decentralized stablecoin project built on the Ethereum blockchain. It aims to create a new kind of stablecoin that is not backed by any external assets like fiat currencies or commodities, but instead by the economic activity of its own ecosystem. The Fei Protocol accomplishes this through a mechanism called direct incentives. When users purchase FEI stablecoins, they receive a proportional amount of the project's governance token, called TRIBE. The more FEI that is purchased, the more TRIBE is minted and distributed to buyers. As the ecosystem grows, the value of the TRIBE token is expected to appreciate, providing stability to the FEI stablecoin. [19]

The Fei protocol uses Rari Fuse pools as a means of providing liquidity to the FEI-TRIBE pool. The Fuse pool acts as a lender, providing capital to the Fei protocol in exchange for interest.

The Fei protocol was hacked on the **30th of April 2022** due to an exploit targeting its underlying Rari Fuse pool, for a total loss of around **80M USD**. The stolen funds were not returned by the attacker. The Fei team launched a compensation program for users who were affected by the attack. This program involved issuing a new token which could be redeemed for its underlying value in ETH once the Fei Protocol treasury had sufficient funds to do so. [20]

The vulnerability exploited in this attack is similar to the "CREAM Finance" attack described above. Both of these contracts are based on the source code of another popular protocol, called Compound Finance. [21]

The interesting point in this case, is that the "Rari Capital" developers tried to patch the CREAM vulnerability by deploying a configurable contract-wide **and** protocol-wide reentrancy guard. While the implementation seems robust, it failed to defend a function of the protocol which ultimately led in the exploit.

The **root cause** of this attack stems from a reentrancy vulnerability which enabled an

attacker to borrow assets and then withdraw his deposited collateral without repaying the loan.

Specifically, the attacker posted multiple tokens as collateral, which allowed him to borrow ETH from the pool. However, in order to transfer the borrowed ETH, the vulnerable contract invoked the adversary's fallback function, which in turn performed an external call to the unprotected `exitMarket` function. This call succeeded, since the borrow records were to be updated **after** the ETH transfer (violating the CEI pattern) and thus the adversary appeared to have no outstanding loans. Therefore, he was able to withdraw his collateral and walk away with the borrowed ETH as profit.

```
1    function borrowInternal(uint borrowAmount) internal nonReentrant(false)
        returns (uint) {
2        // [...]
3        return borrowFresh(msg.sender, borrowAmount);
4    }
5
6    function borrowFresh(address payable borrower, uint borrowAmount) internal
         returns (uint) {
7        // [...]
8
9        /*
10        * We invoke doTransferOut for the borrower and the borrowAmount.
11        *  doTransferOut reverts if anything goes wrong, since we can't be
              sure if side effects occurred.
12        */
13
14       doTransferOut(borrower, borrowAmount);  // VULNERABLE
15
16       /* We write the previously calculated values into storage */
17       accountBorrows[borrower].principal = vars.accountBorrowsNew;
18       accountBorrows[borrower].interestIndex = borrowIndex;
19       totalBorrows = vars.totalBorrowsNew;
20       // [...]
21   }
22
23   function doTransferOut(address payable to, uint amount) internal {
24       // Send the Ether and revert on failure
25       (bool success, ) = to.call.value(amount)("");  // VULN: Reentrancy
26       require(success, "doTransferOut failed");      // attacker calls
             exitMarket()
27   }
28
29  /**
30    * @notice Removes asset from sender's account liquidity calculation
31    * @dev Sender must not have an outstanding borrow balance in the asset,
32    *  or be providing neccessary collateral for an outstanding borrow.
33    */
34   function exitMarket(address cTokenAddress) external returns (uint) {
35       CToken cToken = CToken(cTokenAddress);
36       /* Get sender tokensHeld and amountOwed underlying from the cToken */
37       (uint oErr, uint tokensHeld, uint amountOwed, ) = cToken.
             getAccountSnapshot(msg.sender);  // VULN: Not updated in reentrant
              call
38       // [...]
39   }
```

This attack is **prevented** by securing the `exitMarket` function with the already implemented `nonReentrant` modifier, which disallows calls to **any** protocol function while another

function is executed. Alternatively, following a more robust approach on the CEI pattern, the bookkeeping on the borrowed assets for each user should be performed **before** the transfer takes place, and should be restored if the transfer fails.

5.2 Missing Access Control

The “missing access control” vulnerability class refers to the lack of mechanisms that adequately restrict who can perform certain actions or access certain data within a smart contract. When these mechanisms are not properly implemented, an attacker can exploit this vulnerability to gain unauthorized access to sensitive data or functions within the contract. In Solidity, access control is typically enforced using conditional statements such as `if` and `require`.

Examples of missing access control vulnerabilities include:

- Failure to restrict access to administrative functions: If a smart contract contains functions that can modify the state of the contract, such as adding or removing users, it is important to restrict access to these functions to prevent unauthorized modifications. If access control is not properly implemented, an attacker could potentially modify the contract state in ways that are detrimental to the contract or its users.
- Failure to restrict access to sensitive data: If a smart contract contains sensitive user data, such as personal information or financial data, it is important to restrict access to this data to prevent unauthorized disclosure. If access control is not properly implemented, an attacker could potentially access this data and use it for malicious purposes, breaching the users’ privacy.
- Failure to restrict access to financial transactions: If a smart contract contains financial transactions, it is important to restrict access to these transactions to prevent unauthorized transfers. If access control is not properly implemented, an attacker could potentially transfer funds to an account they control, leading to financial loss for the contract or its users.

To mitigate missing access control vulnerabilities, smart contract developers should carefully review their code and ensure that access control mechanisms are properly implemented and checked. This includes performing rigorous testing and code review to identify and address any potential vulnerabilities before the contract is deployed on the blockchain. Additionally, developers can use established best practices and security frameworks to ensure that their contracts are as secure as possible.

Missing access Control vulnerabilities typically arise due to oversights in the development of the smart contract. In order to minimize the chance of an oversight, a detailed specification of each function’s attributes and intended visibility should be available before the development phase and should be rigorously updated in case of any change. Solidity offers a built-in construct called “function modifier”, that aims to minimize the mistakes made in the implementation of access control mechanisms. The main use case of modifiers is to automatically check a condition prior to executing a function. If the function does not meet the modifier requirement, an exception is thrown, and the function execution stops.

As an example showcasing a “missing access control” vulnerability and its mitigation, suppose that the function `setPrice` is meant to only be called by the owner of the smart con-

tract. Utilising Solidity modifiers, a possible enforcement of this Access Control policy would be the following:

```
contract Bank {
    address private owner;
    uint256 private price;
    constructor () {
        owner = msg.sender;  // Set `owner` to the address of the deployer
    }

    // modifier checks that the caller of the function is the owner
    modifier onlyOwner() {
        require(msg.sender == owner, 'Not Owner');
        _;
    }

    // can be called by anyone due to "public" visibility
    function unsafeSetPrice(uint newPrice) public {
        price = newPrice;
    }

    // only the owner of the contract can call `safeSetPrice`
    // because the `onlyOwner` modifier is specified
    function safeSetPrice(uint newPrice) public onlyOwner {
        price = newPrice;
    }

    // Due to the modifier, the body of `safeSetPrice` becomes:
    //   function safeSetPrice(uint newPrice) public {
    //       require(msg.sender == owner, 'Not Owner');
    //       price = newPrice;
    //   }
}
```

In the case of `safeSetPrice`, Solidity executes the instructions of the modifier, replacing the "_" placeholder with the body of the function. If the condition inside the `require` statement does not hold, the execution of the function is cancelled. Notice that the modifier affecting any function should be included in the function's definition.

Next, we are going to discuss three real-world exploits due to missing access control vulnerabilities.

5.2.1 punk.protocol

The core concept of the Punk Protocol was to provide its users with a high-yield compounding platform, following the traditional pension concept. Users would deposit tokens and set an expiry date, only after which they could "unlock" their deposits and the accumulated interest. [22]

The project was hacked on the **10th of August 2021**, right after its deployment on the Ethereum network, for a total of **8.9M USD**. The smart contracts used by the protocol were not audited before its launch. It's interesting to note that the exploit transaction was **frontran** by a bot. The bot observed the attacker's transaction and performed it first, extracting some of the tokens from the protocol before the attacker could. However, a bug in the bot's code meant that it could only extract some of the tokens targeted by the attacker. In the end, the bot extracted roughly 6M USD from the protocol and another 3M

USD was stolen by the attacker. The owner of the bot returned 5M USD in tokens with the remainder claimed as a finder's fee. [23]

Frontrunning is a technique where an automated process, such as a bot, monitors the Ethereum network for pending transcations, which are transactions that need to be validated before execution. When the process locates a transaction that could yield profit for the bot, it tries to replicate it and submit it to the validation process before the original transaction gets executed. This is possible if, for example, the adversary specifies a higher gas price, so that validators will be incentivized to include the malicious transaction before the legitimate one.

The **root cause** of the attack is a missing an access control check in the `initialize` function of the contract, which sets the address of the external `forge` contract. The `forge` contract is permitted by the application logic to withdraw the funds of the vulnerable contract. Any adversary is able call the publicly visible `initialize` to replace what should have been the protocol's `forgeAddress` with their own malicious contract and withdraw the funds.

```
1     function initialize(
2         address forge_,
3         address token_,
4         address cToken_,
5         address comp_,
6         address comptroller_,
7         address uRouterV2_ ) public {  // VULN: Can be called by anyone with
              custom arguments!
8             addToken( token_ );
9             setForge( forge_ );  // VULN: forge_ is set to be a malicious
                  contract.
10            // [..]
11    }
12
13    function withdrawTo( uint256 amount, address to ) public OnlyForge
          override{
14        uint oldBalance = IERC20( token(0) ).balanceOf( address( this ) );
15        CTokenInterface( _cToken ).redeemUnderlying( amount );
16        uint newBalance = IERC20( token(0) ).balanceOf( address( this ) );
17        require(newBalance.sub( oldBalance ) > 0, "MODEL : REDEEM BALANCE IS
              ZERO");
18        IERC20( token( 0 ) ).safeTransfer( to, newBalance.sub( oldBalance ) );
19        emit Withdraw( amount, forge(), block.timestamp);
20    }
21
22     // @dev This modifier allows only "Forge" to be executed.
23     modifier OnlyForge(){
24        require(_forge == msg.sender, "MODEL : Only Forge");
25        _;
26    }
27
28     // @dev A model must have only one Forge.
29     // IMPORTANT: 'Forge' should be non-replaceable by default.
30    function setForge( address forge_ ) internal returns( bool ){
31        _forge = forge_;
32        return true;
33    }
```

Despite the fact that withdrawal mechanisms are protected by the `OnlyForge` modifier, the `initialize` function had already defined the malicious contract as the `forgeAddress`, and as such, `OnlyForge` did not detect any anomaly, thus authorizing the withdrawal.

Note that in the `setForge` function there is a comment disclosing the developers' intention: "*'Forge' should be non-replaceable by default.*". This behavior, however, is not implemented in code.

Prevention of this attack can be achieved by using a lock in the `setForge` function, so that nobody will be able to call it after the initial setup from the owner. This behavior is consistent with the comment on the source code, and it is what the developers probably intended. Alternatively, `onlyAdmin` or `onlyGovernance` access control modifiers can be used to limit the `initialize` function's scope of permitted callers.

5.2.2 Furucombo

Furucombo is a platform that allows users to bundle various DeFi protocols together into a single transaction. The platform enables users to create custom DeFi strategies by combining different protocols in a way that best suits their needs. Furucombo aims to simplify the DeFi user experience by reducing the number of steps required to execute complex transactions across different protocols. [24]

Furucombo was hacked on the **27th of February 2021** for a total loss of **15M USD**, affecting 22 users. [25]

The **root cause** of this attack stems from an oversight in the access control functionality of a proxy smart contract, which failed to separate the allowed callers of the proxy from the allowed external contracts that the proxy can call. Instead, the lists of permitted callers and callees were unified in the proxy contract. This allowed for a callee to modify the proxy's storage to install the address of a malicious contract. By using the `delegatecall` logic implemented by the proxy, the malicious contract would execute in the context of the Furucombo smart contract, draining its user funds. Since this was a complex attack from a technical perspective, more details can be found in the accompanying repository of this work.

The attack is **prevented** through a whitelist approach. This can be achieved by enforcing the segregation of the allowed callers and the allowed callees, keeping a separate registry for each one, that can only be modified by the owner (or the governing body) of the protocol.

5.2.3 Popsicle Finance

Popsicle Finance is a DEX protocol built on the Ethereum blockchain. Its main feature is that it enables cross-chain swaps between different blockchains, such as Ethereum, Polygon, and Binance Smart Chain. This allows users to easily move assets between these different networks without needing to use a centralized exchange. Popsicle Finance also features automated market-making algorithms to determine token prices and provides liquidity through liquidity pools. [26]

Popsicle Finance was hacked on the **4th of August 2021** for a total loss of over **20M USD**. The attack is classified as a double-claiming attack, which is a loophole of the protocol's reward system that allows the attacker to claim rewards repeatedly. The vulnerable smart contracts were audited before the exploit occurred. [27]

The **root cause** of this attack is an oversight in the reward claim mechanism, which depended on the user's last deposit to calculate the reward amount. However, the mech-

anism did not restrict users without deposits from claiming rewards, which allowed for multiple users to claim the same rewards more than once. The process is as follows: a user first invokes the deposit function to provide liquidity, and gets Popsicle LP tokens (PLP) to be redeemed later. After that, Popsicle Finance will manage the liquidity (interacting with platforms like Uniswap) for the user to make profits. The user can invoke the withdraw function to fetch back the liquidity from Popsicle Finance, which will calculate the amount based on the PLP tokens. The incentive reward comes from the liquidity, which will be accumulated as the time goes by. The user can invoke the `collectFees` function to claim the rewards, which is the key of this attack.

```
1     function collectFees(uint256 amount0, uint256 amount1) external
          nonReentrant updateVault(msg.sender) {
2         // [...]
3     }
4
5     // Function modifier that calls update fees reward function
6     modifier updateVault(address account) {
7         _updateFeesReward(account);  // VULN: Vulnerability exploited here
8         _;
9     }
10
11    // Updates user's fees reward
12    function _updateFeesReward(address account) internal {
13        (uint256 collect0, uint256 collect1) = _earnFees();
14        token0PerShareStored = _tokenPerShare(collect0, token0PerShareStored);
15        token1PerShareStored = _tokenPerShare(collect1, token1PerShareStored);
16
17        if (account != address(0)) {
18            UserInfo storage user = userInfo[msg.sender];
19            user.token0Rewards = _fee0Earned(account, token0PerShareStored);
                  // VULN: Unreasonably large value returned
20            user.token0PerSharePaid = token0PerShareStored;
21            user.token1Rewards = _fee1Earned(account, token1PerShareStored);
22            user.token1PerSharePaid = token1PerShareStored;
23        }
24    }
25
26    // Calculates how much token0 is entitled for a particular user
27    function _fee0Earned(address account, uint256 fee0PerShare_) internal view
          returns (uint256) {
28        UserInfo memory user = userInfo[account];
29        return
30            balanceOf(account)                                    // VULN: user.
                  token0PerSharePaid is 0 on accounts with 0 deposits.
31: 31:           .mul(fee0PerShare_.sub(user.token0PerSharePaid))  // .sub() is
      supposed to subtract a large value, namely the token0PerSharePaid at the
      time of depositing.
32            .unsafeDiv(1e18)                                      // This is
                  actually the first lookup for "account" in "userInfo" map,
                  thus returning 0
33            .add(user.token0Rewards);  // Ultimately, the returned value is
                  the number of PLP shares times the most recent
                  token0PerSharePaid value
34    }
35
36    // Calculates how much token is provided per LP token
37    function _tokenPerShare(uint256 collected, uint256 tokenPerShareStored)
          internal view returns (uint256) {
38        uint _totalSupply = totalSupply();
```

```
39          return   // NOTE: Incremental operation, tokenPerShareStored only
                increases.
40              tokenPerShareStored.add(collected.mul(1e18).unsafeDiv(_totalSupply
                    ));
41      }
```

As it is evident on line 31, the contract depends on `user.token0PerSharePaid`, which is expected to have a large value to restrict the reward amount. However, if the user has zero deposits she will never be registered in the `userInfo` map. Solidity, by default, returns zero on map lookups with unknown keys. Therefore, `user.token0PerSharePaid` is zero and the rewards are way higher than they should be.

This enables an adversary to register a deposit, get LP tokens for her deposit (which are later used to calculate the reward fees), and then transfer these tokens to different accounts with zero deposits and claim reward fees. This process of claiming rewards through different accounts by transferring PLP tokens can be repeated many times, until the pool is drained of its funds.

This attack can be **prevented** by ensuring that the `collectFees` function can only be called by users that have performed at least one deposit. Assuming that on the first deposit the user is registered in the `userInfo` map, a modifier could be created to permit the call only if the `msg.sender` value exists in the map keys.

5.3 Oracle Manipulation

Oracles are services that provide external information to smart contracts on the blockchain. The blockchain is a closed network, and therefore smart contracts running on it do not have access to off-chain data sources or data from other networks. Oracles serve as a bridge between smart contracts and the real world by providing them with access to external data.

Oracles can provide a wide range of data, such as real-world events, market prices and weather reports. They work by fetching data from sources outside of the blockchain and broadcasting it to the smart contract in a format that the contract can understand. Oracles are essential for enabling decentralized applications that rely on external data to function, such as prediction markets, insurance protocols, and decentralized exchanges.

There are two types of oracles: on-chain and off-chain oracles. On-chain oracles are fully integrated within the blockchain network, meaning that they use data available on the blockchain, and provide their output directly to the smart contract. Off-chain oracles, on the other hand, fetch data from external sources and then relay it to the smart contract. A smart contract can “query” the oracle and expect to receive an “answer” to process.

Price oracles in Ethereum are special types of oracles that provide reliable, up-to-date price information for digital assets. They are a crucial component of DeFi applications and are used to enable various financial operations, such as setting the price of assets, triggering certain events, and executing trades. Without price oracles, DeFi applications would not be able to execute actions based on real-world market conditions.

As an example, assume that there is a smart contract that allows users to exchange Ether for a particular token, such as USDC. The smart contract needs to know the current market price of the token, which varies through time, to determine the correct exchange rate. Towards this cause, a price oracle can be used to obtain the current market price of the

token from an off-chain source, such as a centralized exchange, and then use that price to set the exchange rate for the trade on the blockchain.

Chainlink and Uniswap are two popular price oracle solutions in the Ethereum ecosystem that provide reliable price data to smart contracts. Chainlink [28] is an off-chain oracle solution that aggregates data from multiple sources, verifies the accuracy of the data, and delivers it to smart contracts on-chain. Uniswap, on the other hand, uses an on-chain oracle system that relies on the Uniswap Liquidity Pools to provide price data for tokens. The system calculates the price based on the ratio of the reserves of two tokens in the relevant liquidity pool. Uniswap also offers a time-weighted average price (TWAP) oracle to provide a more accurate representation of the current market price. The TWAP oracle calculates the average price of a token over a specified time period by taking the sum of the prices at each time interval and dividing it by the number of intervals.

An "**oracle manipulation**" vulnerability commonly refers to the manipulation of a price oracle used by an external service, so that an adversary can buy or sell a certain asset that is below or above the fair market price on that platform. If the price oracle relies on data stored in the blockchain, there is a possibility that an adversary might be able to influence these data and manipulate the oracle's answer in undesirable ways.

These attacks are typically achieved with the assistance of flash loans. An adversary might request a huge flash loan and use it to affect the ratio of assets in liquidity pools, targeting price oracles that use token balances in their calculations. By manipulating the oracle's answer, a third-party protocol using the price oracle might under-price an asset, allowing the attacker to buy it at a cheaper price than its fair market price.

We will showcase this attack through an example. Suppose that the `PriceOracle` contract represents a price oracle, the `VulnerableDEX` contract represents a decentralized exchange and the `Attacker` contract represents a malicious actor. Assume that we are working with two tokens following the ERC20 standard, and let's name these tokens `WETH` and `USDC` (which are existing tokens, but we are going to use custom rates). The source code of the contracts is given below:

```
contract PriceOracle {
    UniswapV2Pair pool;  // WETH/USDC pool
    // [...]
    // Get price of WETH in terms of USDC.
    function getPriceOfWETH() internal returns (uint) {
        uint supplyWETH, supplyUSDC;
        (supplyWETH, supplyUSDC, ) = pool.getReserves();
        return supplyUSDC / supplyWETH;
    }

    // Get price of USDC in terms of WETH.
    function getPriceOfUSDC() internal returns (uint) {
        uint supplyWETH, supplyUSDC;
        (supplyWETH, supplyUSDC, ) = pool.getReserves();
        return supplyWETH / supplyUSDC;
    }

    function getPrice(address tokenIn, address tokenOut) public returns (uint)
    {
        if (tokenIn == WETH_ADDRESS) {
            return getPriceOfWETH();
        }
        else {
            return getPriceOfUSDC();
```

```
        }
    }
}
```

The `PriceOracle` contract has one public function, called `getPrice(address tokenIn, address tokenOut)`. This function returns the price of `tokenIn` in terms of `tokenOut`. The calculation is based on the supply of each token in a Uniswap Liquidity Pool. For example, if there are 200 `WETH` and $10,000$ `USDC` in the pool, the price of one `WETH` is 50 `USDC` and the price of one `USDC` is 0.02 `WETH`.

```
contract VulnerableDEX {
    // [...]
    // Exchange `amountIn` of `tokenIn` for the corresponding
    // amount of `tokenOut` based on the price oracle's answer.
    function exchange(address tokenIn, address tokenOut, uint amountIn) {
        uint exchangeRate =  PriceOracle(ORACLE_ADDRESS).getPrice(tokenIn,
            tokenOut);
        uint amountOut = exchangeRate * amountIn;

        // Transfer `amountIn` of `tokenIn` from sender to this contract.
        bool success = ERC20(tokenIn).transferFrom(msg.sender, address(this),
            amount);
        require(success, "Sender does not have enough funds of token A.");

        // Transfer `amountOut` of `tokenOut` from this contract to sender.
        ERC20(tokenOut).transfer(msg.sender, amountOut);
    }
}
```

The `VulnerableDEX` contract has one public function, called `exchange(address tokenIn, address tokenOut, uint amountIn)`. This function attempts to exchange `amountIn` of `tokenIn` for the corresponding amount of `tokenOut`, based on the exchange rate obtained through the price oracle. Assuming the balances of the previous example, `exchange(WETH, USDC, 10)` means that the sender exchanges 10 `WETH` to the DEX and receives 500 `USDC`.

```
contract Attacker {
    VulnerableDEX dex;
    // [...]
    function getFlashLoan(address token, uint amount) {
        // Get a flash loan of `amount` tokens of `token`
    }

    function repayFlashLoan(address token, uint amount) {
        // Repay a flash loan of `amount` tokens of `token`
    }

    function attack(address WETH_ADDR, address USDC_ADDR) {
        dex.exchange(USDC_ADDR, WETH_ADDR, 1e2);
        getFlashLoan(USDC_ADDR, 1e5);
        pool.exchange(USDC_ADDR, WETH_ADDR, 1e5);
        dex.exchange(WETH_ADDR, USDC_ADDR, 198);
        repayFlashLoan(USDC_ADDR, 1e5);
    }
}
```

The `Attacker` contract is the most complicated one, since it uses flash loans. Flash loans are a bit complex from a developer point of view, so we omit the implementation of the relevant functions from this text. The full code snippet can be found in the accompanying repository.

The interesting function performing the exploit is named `attack`. Let's assume the balances of the previous example, namely that the Uniswap Liquidity Pool has 200 `WETH` and $10,000$ `USDC`. Assume that the attacker has an initial capital of 100 `USDC`, and the DEX contract owns $2,000$ `WETH` and $100,000$ `USDC`.

The steps of the `attack` function when executed line by line, are the following:

1. The attacker calls the DEX to exchange his initial capital of 100 `USDC`. The DEX consults the price oracle which reports a price of 0.02 `USDC` per `WETH`. Therefore the attacker obtains 2 `WETH` from the DEX and is left with 0 `USDC`. The Liquidity Pool balance is left intact, and the DEX now has $1,998$ `WETH` and $100,100$ `USDC`.

2. The attacker gets $10,000$ `USDC` through a flash loan, which he will have to repay later. The attacker is in possession of 2 `WETH` and $10,000$ `USDC`.

3. The attacker now contacts **directly** the Uniswap Liquidity Pool to request an exchange of his $10,000$ `USDC` for `WETH`. Assume that the Liquidity Pool uses a similar Price Oracle with the DEX, but since it cannot be empty of its `WETH` reserve, it returns back 150 `WETH`. Therefore, the attacker now has 152 `WETH` and 0 `USDC`, the DEX balances are left intact, and the Uniswap pool is left with 50 `WETH` and $20,000$ `USDC`. In this situation we say that the pool is "**tilted**", meaning that it contains disproportionate amounts of each token.

4. The attacker now requests an exchange of his 152 `WETH` for `USDC` from the DEX. The DEX consults the price oracle, which now inevitably returns an overpriced result. The function `getPriceOfWETH` is called, which calculates the price of `WETH` to be $20,000/50 = 400$ `USDC` per `WETH`, based on the Uniswap LP balances. When compared to the initial reported price of 50 `USDC` per `WETH` in the beginning of this transaction, the price is now evaluated 8 times higher. Therefore, the DEX takes 152 `WETH` and returns $152 * 400 = 60,800$ `USDC`. The attacker now has 0 `WETH` and $60,800$ `USDC`. The DEX has $2,150$ `WETH` and $39,200$ `USDC` and the LP balances are left intact.

5. Lastly, the attacker repays the flash loan returning $10,000$ `USDC` and keeping $29,200$, for a total profit of $29,100$ `USDC` since the beginning of the attack.

Oracle manipulation threats are inherent in on-chain oracles, since third-party data stored on the blockchain might change frequently and in unexpected ways. Therefore, smart contracts that rely on on-chain price oracles should calculate the desired values by combining many different price feeds, that obtain their results from various DEXs. This does not mitigate the issue completely, but offers a more robust approach when relying on this type of oracles. A completely different approach would be to consider an off-chain, centralized price oracle, which is immune to on-chain manipulation, but shifts the trust boundaries to the central authority controlling the oracle.

Next, we are going to discuss two real-world exploits due to oracle manipulation vulnerabilities. Since this class of vulnerabilities usually involves many protocols and subtle bugs in the price calculations, we will not inspect the source code of the examined smart contracts. The interested reader shall refer to the accompanying repository for the precise code snippets introducing the vulnerability in each case.

5.3.1 Inverse Finance

Inverse Finance is a DeFi protocol built on the Ethereum blockchain that offers users a decentralized platform for swapping stablecoins, trading on margin, and providing liquidity to various liquidity pools. The protocol aims to provide a range of financial services similar to traditional finance, such as synthetic assets, leverage, and yield farming, but in a decentralized manner. Inverse Finance uses a governance token, INV, to enable its users to vote on proposals and changes to the protocol. It was launched in late 2020.

The protocol was hacked on the **16th of June 2022** for a total loss of **7M USD**, out of which 1.2M were obtained by the attacker and laundered through *Tornado.Cash*. It is worth noting that the exploit we describe here is the second hack of the protocol, after a vulnerability worth 15M was exploited two months earlier. [29]

Inverse Finance uses an on-chain oracle to calculate the spot price of the `yvCurve-3crypto` token. The **root cause** of the attack is that the oracle uses the raw balances of `WBTC`, `WETH` and `USDT` tokens owned by the `crv3crypto Liquidity Pool` directly in its calculation. The attacker performed a flash-loan assisted exchange in the `crv3crypto LP` swapping `WBTC` for `USDT`, thus tilting the pool in favor of `WBTC`. Apparently, the correlation between `WBTC` and `USDT` in the price calculation was flawed: the increment in `WBTC` had a greater impact than the decrement in `USDT`, which resulted in a much higher price answer.

The adversary utilized the bug in the calculation logic to first post `yvCurve-3crypto` as collateral to Inverse Finance, and then manipulate the oracle to evaluate the token in a much higher price than the one used at the time of posting. As a result, he was able to borrow an excessive amount of Inverse's token, as if it was backed by double the original collateral, therefore leading to a profit.

Building resilient and tamper-proof on-chain price oracles to **prevent** similar attacks requires precise calculations in order to weight the contribution of each different token to obtain the final value. Therefore, thorough testing needs to be performed on the formulas employed by the oracles, taking into consideration edge-case and situations arising from malicious flash-loan assisted exchanges.

5.3.2 Value DeFi

Value DeFi is a decentralized finance protocol that aims to provide users with a suite of tools and products for trading, lending, and earning interest on their cryptocurrency holdings. The protocol is built on the Ethereum blockchain and features a number of innovative features, including a liquidity pool that automatically rebalances to provide optimal returns and an insurance fund to protect users against losses due to unforeseen events. Value DeFi also offers a governance token, which allows users to participate in the decision-making process of the platform.

The Value DeFi smart contracts responsible for the protocol vaults where hacked on the **14th of November 2020** for a total loss of **7M USD**. The exploit targeted the second version of the smart contracts, which had not been audited. [30]

Value DeFi uses an on-chain oracle to calculate the exchange rate of its `mvUSD` token in terms of the `3crv` token. The `3crv` is an LP token employed by Curve's `3pool`, which is a Liquidity Pool managing 3 USD-backed assets: `USDC`, `USDT` and `DAI`. When a deposit consisting one of the three assets is made, the pool rewards the depositor with `3crv` tokens, which they can later redeem to withdraw an asset of their choice from these three.

The **root cause** of the exploit can be located in the price oracle logic, which uses the USDC spot price of 3pool to calculate the conversion rate of two other tokens, BCrv and CCrv, to 3crv. It does that by converting BCrv and CCrv to USDC and then calculates the 3Crv as if the amount of obtained USDC was deposited in the 3pool.

The attacker used flash loans to drain the 3pool of its USDC. After that, the oracle would convert BCrv and CCrv to USDC with the usual BCrv per USDC and CCrv per USDC convertion rates, but when it tried to deposit USDC in the 3pool, the amount of rewarded 3crv tokens would be much higher, since the 3pool was drained of its USDC. As a result the 3crv per USDC price was much larger, in order to incentivize deposits of USDC. This resulted in BCrv and CCrv to be considered momentarily overvalued by Value DeFi, and thus were converted to more 3crv tokens than they normally whould.

The attacker managed to mint 25M mvUSD tokens, by depositing to Value, which were worth 25M 3crv at the time of the deposit. He then manipulated the 3pool USDC supply and withdrew his 25M mvUSD for 33M 3crv, which he then converted to stablecoins.

This attack would probably be **prevented** if the on-chain oracle used the convertion rates of all three assets in the 3pool, instead of just USDC. Since they are all USD-backed, meaning that the asset per US Dollar rate is close to 1, no special weights need to be assigned, as long as all three assets participate equally in the calculation.

5.4 Insufficient Data Validation

Insufficient data validation is a vulnerability class in smart contracts where input data is not properly validated before being processed. Functions that accept invalid input parameters or do not properly check the format or type of input data might be exploitable.

Vulnerabilities of this class can lead to a variety of attacks, including:

- Integer overflow/underflow: If the contract does not properly validate input data and an attacker can supply a large or small integer value that exceeds the maximum or minimum allowed value, it could cause an overflow or underflow. This can lead to unexpected behavior or even allow an attacker to gain control of the contract.
- Denial of Service (DoS): An attacker can send unexpected input data to the contract which could cause it to enter an infinite loop or crash, causing a DoS attack. This could render the contract unresponsive to legitimate users and prevent them from using the contract.
- Malicious function calls: If input data is not validated, an attacker could call a function with malicious intent, such as changing ownership of the contract, altering the contract state or stealing funds.
- Arbitrary code execution: If input data such as external addresses to be called, are not properly validated, an attacker could supply code to be executed in the context of the vulnerable contract or take over the transaction's control flow.

All of these attacks can result in significant financial losses or even to a complete takeover of the vulnerable contract. In order to prevent exploitation, it is important to perform adequate data and type validation on both input and output data, as well as on intermediate variables. Towards this cause, it is recommended to use open-source libraries and built-in Solidity functions that provide secure data manipulation and arithmetic operations. A

whitelist approach is recommended when the contract is required to perform calls to external addresses.

Next, we are going to discuss four real-world exploits due to insufficient data validation vulnerabilities.

5.4.1 TempleDAO

TempleDAO is a decentralized autonomous organization built on the Ethereum blockchain. Its platform allows users to create and participate in on-chain proposals, enabling the community to collectively make decisions about the future direction and development of the project.

TempleDAO offers a liquidity pool called “STAX”. The smart contract implementing this pool was hacked on the **11th of October 2022** for a total loss of **2.3M USD**. [31]

The **root cause** of the attack is a lack of verification on whether expected funds from a transfer were actually received. The issue was caused due to an upgrade in the “STAX” smart contracts. Since the smart contracts in Ethereum are immutable, TempleDAO issued a new smart contract implementing the “STAX” liquidity pool. This new smart contract offered users the ability to move (“migrate”) their staked capital from the old smart contract to the new one. This is done by invoking the `migrateStake` function on the new contract, which calls the `migrateWithdraw` function of the old contract in order to transfer the funds to the newest version. The new contract assumes that the transfer always succeeds and credits the sender with the requested amount. Therefore, an adversary is able to request the migration of a huge amount of tokens, even without actually possessing the funds. The `migrateStake` function will try to withdraw the funds from the old contract. This transfer will fail but will not revert (it won’t be cancelled) and the function logic will continue to the next step, which credits the requesting address with the requested amount of tokens in the new contract. The adversary is then able to redeem the new tokens she just obtained and walk away with the profit.

```
1    function migrateStake(address oldStaking, uint256 amount) external {
2        StaxLPStaking(oldStaking).migrateWithdraw(msg.sender, amount);  //
             VULN: Caller expects `amount` tokens to be transfered to them -
             does not verify it
3        _applyStake(msg.sender, amount);  // Credits `amount` tokens to sender
              anyway
4    }
5
6    function _applyStake(address _for, uint256 _amount) internal updateReward(
         _for) {
7        _totalSupply += _amount;
8        _balances[_for] += _amount;
9        emit Staked(_for, _amount);
10   }
```

The attack is effectively **prevented** by verifying that the expected funds are actually received. This can be done either by checking the return value of the `migrateWithdraw` function, or by comparing the token balance of the contract right before and right after the transfer call is executed. If the balance remains the same, it is safe to assume that the transfer failed.

5.4.2 DeFi Saver

DeFi Saver is a platform that allows users to manage their DeFi assets from a single interface. It provides various tools and services, such as automated trading bots, collateral swapping, and token swapping. DeFi Saver aims to simplify the process of managing DeFi assets and help users optimize their returns while minimizing risks.

A vulnerability in the DeFi Saver smart contracts was identified on the **5th of January 2021** by independent researchers. The DeFi Saver admins were informed and took action to mitigate the issue before a malicious actor could exploit it. The potential funds at risk were **3.5M USD**. [32]

The **root cause** of this attack is insufficient protection against hostile user input. An adversary could exploit a subtle flaw in the contract logic, which allowed him to obtain collateral assets and outstanding loans of other users. The ability to specify a target user and a custom malicious proxy contract without restrictions, enabled the adversary to perform this attack.

```
1    /// @notice Called by Aave when sending back the FL amount
2    function executeOperation(address _reserve, uint256 _amount, uint256 _fee,
         bytes calldata _params)
3    external override {
4        (address cCollateralToken, address cBorrowToken, address user, address
             proxy)
5        = abi.decode(_params, (address,address,address,address));
6
7        // repay compound debt
8        require(CTokenInterface(cBorrowToken).repayBorrowBehalf(user, uint(-1)
             ) == 0, "Repay borrow behalf fail");
9
10       // transfer cTokens to proxy  // VULN: proxy is attacker-controlled
11       uint cTokenBalance = CTokenInterface(cCollateralToken).balanceOf(user)
             ;
12       require(CTokenInterface(cCollateralToken).transferFrom(user, proxy,
             cTokenBalance));
13
14       // borrow  // VULN: Imposes a loan-debt on the attacker-controlled
             proxy
15       bytes memory proxyData = getProxyData(cCollateralToken, cBorrowToken,
             _reserve, (_amount + _fee));
16       DSProxyInterface(proxy).execute(COMPOUND_BORROW_PROXY, proxyData);
17
18       // Repay the flash loan with the money DSProxy sent back
19       transferFundsBackToPoolInternal(_reserve, _amount.add(_fee));
20   }
```

It is worth mentioning that the attack requires two dynamic conditions to be satisfied in order to succeed. First, the victim user needs to have an approval of his collateral token to the vulnerable contract which is greater or equal to his balance. This condition enables the vulnerable contract to transfer the user's funds to the malicious proxy contract. Second, the victim user needs to have outstanding loans that are less than the collateral he provided, otherwise the attack would not be profitable for the attacker, who essentially repays the victim's loans to redeem his collateral assets. Due to its technical nature, more information on this attack can be found in the accompanying repository.

In order to **prevent** this attack, the vulnerable contract should restrict the ability of an adversary to specify user/proxy pairs. This could be implemented by enforcing users to

register a proxy of their choice when using the smart contract for the first time. Subsequent interactions with the smart contract would use the registered proxy by default, thus stripping the attacker of the ability to migrate the funds to his custom proxy contract. In order to ensure that user's funds are not manipulated against his will, the user should limit or zero out the allowance of his collateral to the vulnerable contract, after performing the desired transaction.

5.4.3 ForceDAO

The Force DAO is a decentralized autonomous organization built on the Ethereum blockchain that aims to provide yield farming opportunities and rewards to its users. Force DAO's strategy involves investing in stablecoin pools and automated market makers on DEXs to generate returns for its users.

A critical vulnerability in the ForceDAO smart contracts was exploited on the **4th of April 2021**, by a whitehat hacker. The whitehat hacker withdrew **9.6M USD**, which he later returned to the protocol. However, 250K USD were also stolen from blackhat hackers that noticed and replicated the whitehat exploit. [33]

The **root cause** of this attack is a missing check on the return value indicating whether a transfer was successful or not. In the case of success, the contract should mint tokens to the user that requested the transfer. In the case of failure the function should revert. With no checks performed, an adversary could request a transfer of an arbitrary large amount of tokens without actually owning them, the transfer would fail, but the corresponding amount of minted tokens would be credited to his account, as if the transfer was successful. The adversary could later redeem the minted tokens for actual tokens and swap them for ETH.

```
1  // xFORCE
2  function deposit(uint256 amount) external nonReentrant {
3      uint256 totalForce = force.balanceOf(address(this));
4      uint256 totalShares = totalSupply();
5
6      if (totalShares == 0 || totalForce == 0) {
7          _mint(msg.sender, amount);
8      }
9      // Calculate and mint the amount of xForce the Force is worth. The ratio
           will change
10     // overtime, as xForce is burned/minted and Force deposited + gained from
           fees / withdrawn.
11     else {
12         uint256 what = amount.mul(totalShares).div(totalForce);
13         _mint(msg.sender, what);  // VULN: xFORCE minted before transfer
14     }
15     // Lock the Force in the contract
16     // VULN: Does *not* revert if `transferFrom` fails
17     force.transferFrom(msg.sender, address(this), amount);
18 }
19
20 // FORCE -- invoked from "force.transferFrom()"
21 function transferFrom(address _from, address _to, uint256 _amount) public
       returns (bool success) {
22     // [..]
23     // VULN : Might not revert if unsuccessful
24     return doTransfer(_from, _to, _amount);
25 }
26
```

```
27 // Called by "force.transferFrom()"
28 function doTransfer(address _from, address _to, uint _amount) internal returns
       (bool) {
29     if (_amount == 0) {
30         return true;
31     }
32     require((_to != 0) && (_to != address(this)));
33     // If the amount being transfered is more than the balance of the
34     //  account the transfer returns false
35     var previousBalanceFrom = balanceOfAt(_from, block.number);
36     if (previousBalanceFrom < _amount) {
37         return false;  // VULN: Does not revert, returns just false
38     }
39     // [..]
40     return true;
41 }
```

The vulnerability is **mitigated** if the return value of `force.transferFrom()` is verified by the `deposit` function to be `true`. If this is not the case, the function should explicitly use the `revert` statement to cancel the transaction.

5.4.4 MonoSwap

Monoswap is a DEX that operates on the Ethereum blockchain. It is designed to provide users with a simple and efficient platform to trade Ethereum-based tokens. The platform uses an AMM model, where trades are executed by smart contracts that hold liquidity pools of tokens.

The Monoswap smart contracts were exploited on the **30th of November 2021** for a total loss of **30M USD**, after just one month of operation. It is worth noting that Monoswap had undergone 3 security audits before deployment. [34]

The **root cause** of the vulnerability is that the contract permitted the exchange of a token with itself in a single trade. The adversary targeted the native token of the contract, MONO, since its price was also determined by the contract's state. When a trade happens, the pool receives an amount of the input token, which leads to greater supply of this token and thus lower price. The reverse is true for the output token, since the pool gives away a portion of this token, leading to lower supply and thus higher price. The price updates resulting from a swap of an input and an output token were independently and sequentially processed by the contract, with the input token price calculation preceding the output token. By continuously swapping MONO with itself, the adversary was able to pump its price, since any change in the price made by the input token calculation did not have any effect due to the output token calculation that was performed last. Using his overpriced MONO tokens, the adversary was able to purchase all other assets in the pool and effectively deplete it of its funds.

```
1 function swapExactTokenForToken(address tokenIn, address tokenOut, uint
      amountIn, uint amountOutMin, address to, uint deadline)
2 external virtual ensure(deadline) returns (uint amountOut) {
3     amountOut = swapIn(tokenIn, tokenOut, msg.sender, to, amountIn); // VULN
4     require(amountOut >= amountOutMin, 'MonoX:INSUFF_OUTPUT');  // tokenIn ==
          tokenOut
5 }
6
7 // swap from tokenIn to tokenOut with fixed tokenIn amount.
```

```
8  function swapIn (address tokenIn, address tokenOut, address from, address to,
       uint256 amountIn) internal lockToken(tokenIn)
9  returns(uint256 amountOut)  {
10     address monoXPoolLocal = address(monoXPool);
11     uint256 oneSideFeesInVcash = tokenInPrice.mul(amountIn.mul(fees)/2e5)/1e18
           ;
12
13     amountIn = transferAndCheck(from,monoXPoolLocal,tokenIn,amountIn);
14     (uint tokenInPrice, uint tokenOutPrice, amountOut, uint tradeVcashValue) =
           getAmountOut(tokenIn, tokenOut, amountIn);
15
16     // trading in  // VULN: Price of tokenIn decreases
17     _updateTokenInfo(tokenIn, tokenInPrice, 0, tradeVcashValue.add(
           oneSideFeesInVcash), 0);
18
19     // trading out  // VULN: Price of tokenOut increases
20     if (to != monoXPoolLocal) {
21       IMonoXPool(monoXPoolLocal).safeTransferERC20Token(tokenOut, to,
             amountOut);
22     }
23     updateTokenInfo(tokenOut, tokenOutPrice, tradeVcashValue.add(
           oneSideFeesInVcash), 0,
24     to == monoXPoolLocal ? amountOut : 0);
25
26     // VULN: Since tokenIn == tokenOut and the increase discards the decrease
           completely, MONO price is pumped up
27     // [..]
28 }
```

The vulnerability is effectively **prevented** by verifying that the addresses of the input and the output tokens in the `swapExactTokenForToken` function are different. If this is not the case, the function should explicitly use the `revert` statement to cancel the transaction. This pattern should be implemented in every function of the protocol that performs token swaps.

6. CONCLUSIONS AND FUTURE WORK

Through our research, we inspected the principle ideas and core components behind the blockchain technology. We used the Ethereum blockchain as a real-world implementation of the technology and examined its unique aspects, including its powerful computational platform in the form of smart contracts. We delved into the most popular and widely used services built on top of the Ethereum network, explaining some novel concepts realized through the blockchain technology. At the core of this work, we shifted our focus in the security aspect of smart contracts. We examined twelve real-world attacks targeting the source code of various smart contracts, and we performed a thorough vulnerability analysis to uncover the root causes that lead to their exploitation. We were able to split and classify the attacks based on the exploited vulnerability in each case. The vulnerability classes we examined in the course of this work are the following:

- Reentrancy: We concluded that **three** attacks stem from a reentrancy vulnerability. This is a novel threat in the landscape of classic software assessment, introduced by the nature of the blockchain and the Contract-Oriented programming paradigm. Vulnerabilities of this class are usually hard to spot but relatively easy to exploit once identified.
- Missing Access Control: We concluded that **three** attacks originate from a missing access control vulnerability. This is a classic class of vulnerabilities, easy to exploit once spotted by a mindful adversary.
- Oracle Manipulation: We concluded that **two** attacks derive from an oracle manipulation scheme. These vulnerabilities are also novel, introduced by the closed nature of a blockchain system. Their exploitation is usually hard, and requires good understanding of financial concepts.
- Insufficient Data Validation: We concluded that **four** attacks are due to insufficient data validation practices. This is also a common threat in software assessment, highlighting the need for developers to follow standard coding practices and perform thorough audits to every smart contract before deployment.

For every vulnerability class and for each specific attack, we provided mitigation recommendations and prevention techniques, based on standard coding practices. Through our work we hope to enhance the security awareness of developers in the smart contract space and contribute towards the development of secure software that the users will trust, leading to faster and wider adoption of the blockchain technology.

Moving forward, we aim to enhance the accompanying public repository with more real-world hacks, showcasing the various and creative ways that the examined vulnerabilities can be exploited in a smart contract. Moreover, we hope to examine different vulnerability classes, adding to the ones presented in this work, so we can obtain a broader view of the current landscape in smart contract security.

ABBREVIATIONS - ACRONYMS

AMM	Automated Market Maker
CEI	Checks-Effects-Interactions
CEX	Centralised Exchange
CPMM	Constant-Product Market Maker
DAO	Decentralized Autonomous Organization
DeFi	Decentralised Finance
DEX	Decentralised Exchange
EOA	Externally Owned Account
ERC	Ethereum Request for Comments
ETH	Ether
EVM	Ethereum Virtual Machine
FL	Flash Loan
LP	Liquidity Pool
PoC	Proof of Concept
PoS	Proof-of-Stake
PoW	Proof-of-Work
USD	U.S. Dollar
VULN	Vulnerability

BIBLIOGRAPHY

[1] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[2] V. Buterin. Ethereum white paper: A next generation smart contract and decentralized application platform. 2014.

[3] D. Karakostas. https://github.com/blockchain-technology-lab/bdl-course.

[4] G. Wood A. M. Antonopoulos. In *Mastering Ethereum*, 2018.

[5] A. M. Turing. On computable numbers, with an application to the entschei- dungsproblem. In *Proceedings of the London mathematical society*, page 2(1):230–265, 1937.

[6] https://docs.soliditylang.org/.

[7] https://eips.ethereum.org/eips/eip-20.

[8] https://eips.ethereum.org/eips/eip-721.

[9] https://docs.uniswap.org/.

[10] https://resources.curve.fi/base-features/understanding-curve.

[11] https://docs.aave.com/hub/.

[12] https://github.com/tornadocash/docs.

[13] https://www.akropolis.io/.

[14] https://cryptobriefing.com/defi-project-akropolis-lost-2-million-heres-what-theyre-doing-about-it/.

[15] https://rekt.news/akropolis-rekt/.

[16] https://cream.finance/.

[17] https://rekt.news/cream-rekt/.

[18] https://medium.com/cream-finance/c-r-e-a-m-finance-post-mortem-amp-exploit-6ceb20a630c5.

[19] https://learn.bybit.com/crypto/fei-protocol/.

[20] https://rekt.news/fei-rari-rekt/.

[21] https://docs.compound.finance/.

[22] https://www.halborn.com/blog/post/explained-the-punk-protocol-hack-august-2021.

[23] https://rekt.news/punkprotocol-rekt/.

[24] https://medium.com/furucombo/furucombo-post-mortem-march-2021-ad19afd415e.

[25] https://rekt.news/furucombo-rekt/.

[26] https://blocksecteam.medium.com/the-analysis-of-the-popsicle-finance-security-incident-9d9d5a3045c1.

[27] https://rekt.news/popsicle-rekt/.

[28] https://docs.chain.link/.

[29] https://rekt.news/inverse-rekt2/.

[30] https://rekt.news/value-defi-rekt/.

[31] https://rekt.news/templedao-rekt/.

[32] https://media.dedaub.com/ethereum-pawn-stars-5-7m-in-hard-assets-best-i-can-do-is-2-3m-b93604be503e.

[33] https://rekt.news/force-rekt/.

[34] https://medium.com/monoswap/exploit-post-mortem-33921a779b43.