**Ethereum Smart Contract Security**

**What are the existing vulnerabilities in Ethereum Smart Contracts?**

There are several existing vulnerabilities in Ethereum smart contracts, some of which are:

**01. Indirect execution of unknown code**

The possibility of indirect execution is there due to the presence of the fallback function feature in smart contracts. There are several reasons why this function can be called:

* Calling a function of another contract using ABI. If there is a typo in the signature string passed for encoding, or a function with such a signature does not exist, then the fallback function will be called.
* Deposit to another contract generates a call to its fallback function.
* Calling a function of another contract using the API. If the developer makes a mistake when declaring the interface of the called contract (for example, mixes up the type of a parameter), the fallback function will be called.

**02. Reentrancy**

Reentrancy is one of the common Ethereum smart contract vulnerabilities.

The thing is that in Ethereum, calls to functions of other smart contracts occur synchronously. That is, the calling code waits for the end of the execution of the external method before continuing its own work. This can cause the called contract to use the intermediate state of the calling contract.

This situation is not always obvious during the development process if all the possible fraudulent actions on the part of the called contract are not taken into account.

**03. Incorrect calculation of the output token amount**

Another smart contract weakness can be found in some modern DeFi smart contracts that deal with enormous amounts of money depicted in tokens or ETH value.

In such protocols, a lot of operations in contract logic are connected with tokens’ transfers to and from the contract. It creates a wide field for different mistakes connected to correct percentages, fees, and profits calculations. That’s why it’s important to have reliable [token development company](https://blaize.tech/token-development-services/) to help ensure the success of your project.

The most common errors include the following:

* incorrect decimals handling, especially when dealing with such a token as USDT;
* incorrect order of operation during fees calculations, which leads to a significant  accuracy loss;
* the accuracy constant that was actually forgotten in the math operations.

**04. Interface/naming issues**

An example of these smart contract risks is the Rubixi smart contract.

Rubixi was a Ponzi game, in which the owner could transfer the fees accumulated in the financial pyramid to themselves.

Usually, the owner is set in the constructor of the contract, which is called when it is created. The same logic was implemented in the Rubixi’s smart contract.

It should be noted that in the Solidity versions prior to 0.4.22, constructors were defined as functions with the same name as the name of the contract. At some point, the contract was renamed from Dynamic Pyramid to Rubixi, but the developer forgot to change the name of the constructor. This way, anyone who called the Dynamic Pyramid function could become the owner of a contract and steal the accumulated funds.

**05. Dependency on the order of execution**

The state of the contract is determined by the values ​​of its variables, which are changed by calling its functions. Calling a smart contract function is the same transaction as a transaction of ETH or ERC-20 token transfer. These transactions are finalized by the network only after the next block creation is complete.

Thus, when the user sends a transaction to call a contract function, they cannot be sure that the transaction will be executed in the same state of the contract in which it was at the time of sending. This can happen because other transactions in the same block have changed the state of the contract.

Moreover, miners have some freedom in ordering transactions when forming a block, as well as in choosing to include a particular transaction in a block. In some cases, the impossibility of determining the state of the contract, in which the transaction will be executed, can cause another smart contract weakness.

It also becomes especially dangerous to interact with contracts written in such a way that their behavior can be changed over time.

**06. Time component**

The next issue in our smart contract vulnerabilities list is the time component. Sometimes, the logic of smart contracts can be time-dependent. The time for a contract is only available in the context of a transaction. The timestamp of a transaction, in turn, is equal to the label of the block that it is included in. Thus, consistency with the state of a smart contract is achieved.

However, this also creates an opportunity for the miner to abuse their position due to some freedom in setting a timestamp for the block. So, the miner has some advantage over other parties to a contract that they could exploit to their own benefit.

**07. Using the blockhash function**

Using the blockhash function is a way of [hacking smart contracts](https://blaize.tech/article-type/defi-security-how-to-prevent-your-defi-project-from-hacking/) similar to the timestamp dependency. It is not recommended to use it for important components for the same reason as the timestamp dependency because miners can manipulate these functions and change the withdrawal of funds in their own favor. This is especially noticeable when the block hash is used as a source of randomness.

**08. Incorrectly handled exceptions**

There are many situations where an exception can be thrown in Solidity, but the way these exceptions are handled is not always the same. Exception handling is based on interactions between contracts. Thus, contracts are vulnerable to attacks from malicious users, and if these exceptions are not handled properly, transactions will be rolled back.

**09. Incorrect work with ERC20 token**

There is a well-known OpenZeppelin implementation of the ERC-20 token that is overused in modern protocols. In most cases, it is completely applicable, and its functionality is enough for correct financial operations. Yet, there is a place for custom implementations of a token standard. Thus, it creates a place for discrepancies between the newly created token and actual ERC20 standard – small inconsistencies like missing return value in transfer() function.

**What are the leading root causes of vulnerabilities in Ethereum Smart Contracts?**

The leading root causes of vulnerabilities in Ethereum smart contracts include:

* Lack of Proper Design: Smart contract developers must have a proper understanding of the problem they are trying to solve and the requirements of the smart contract. Failure to design the contract correctly can result in security vulnerabilities that can be exploited by attackers.
* Lack of Testing: Smart contract testing is crucial to identify vulnerabilities before deploying the contract on the blockchain. Failing to test the contract thoroughly can lead to unexpected behavior or vulnerabilities that can be exploited.
* Misunderstanding of Solidity Language: Solidity is the programming language used to write smart contracts on the Ethereum blockchain. Lack of understanding of Solidity's features and limitations can result in unexpected behavior, errors, and vulnerabilities.
* Incorrect Use of Libraries and APIs: Smart contracts often use libraries and APIs to perform specific tasks. Using these resources without proper understanding or validation can result in vulnerabilities that can be exploited.
* Insufficient Validation of Inputs: Input validation is essential to ensure that the smart contract executes as intended. Failure to validate inputs can lead to unexpected behavior, data corruption, and security vulnerabilities.
* Lack of Awareness of Known Vulnerabilities: Smart contract developers should be aware of known vulnerabilities and how to avoid them. Failure to consider these known vulnerabilities can result in security issues that could have been prevented.

**What are the sub-causes of vulnerabilities in Ethereum Smart Contracts?**

The sub-causes of vulnerabilities in Ethereum smart contracts can be categorized into several areas:

* Coding errors: This includes mistakes in coding the smart contract, such as logic errors, missing or incomplete input validation, or insufficient error handling.
* Design issues: This includes problems with the overall design of the smart contract, such as improper use of variables, data structures, or functions.
* External dependencies: This includes vulnerabilities in external dependencies such as libraries, APIs, or other smart contracts that the smart contract depends on.
* Blockchain specific vulnerabilities: This includes vulnerabilities that are unique to the blockchain environment, such as reentrancy attacks or front-running attacks.
* Human factors: This includes mistakes or oversights made by developers or other humans involved in the development, testing, and deployment of the smart contract.
* Lack of awareness: This includes a lack of awareness of known vulnerabilities and best practices for developing secure smart contracts.
* Compiler and interpreter vulnerabilities: This includes vulnerabilities in the compiler or interpreter used to compile or execute the smart contract.
* Network-related vulnerabilities: This includes vulnerabilities related to network communications, such as man-in-the-middle attacks or denial-of-service attacks.

**What are the detection tools for vulnerabilities in Ethereum Smart Contracts?**

There are several detection tools available for identifying vulnerabilities in Ethereum smart contracts. These tools use different approaches to analyze smart contract code and identify potential security issues. Some of the popular detection tools are:

* Mythril: This is an open-source security analysis tool for Ethereum smart contracts. Mythril uses symbolic execution to detect potential vulnerabilities such as reentrancy attacks, integer overflow, and underflow, and other common security issues.
* Securify: Securify is another open-source security analysis tool that uses a formal verification approach to identify potential security vulnerabilities. It detects a wide ange of vulnerabilities, including reentrancy attacks, transaction-ordering dependence, and more.
* Oyente: Oyente is a tool for analyzing smart contracts that use the Ethereum Virtual Machine (EVM). It detects several types of vulnerabilities such as reentrancy attacks, gas-related vulnerabilities, and more.
* Slither: Slither is an open-source static analysis tool that uses a rule-based approach to detect potential security vulnerabilities in Ethereum smart contracts. It can identify issues such as unused variables, missing input validation, and more.
* Echidna: Echidna is a property-based testing tool for Ethereum smart contracts. It generates random inputs and tests the smart contract's behavior to identify potential vulnerabilities.
* Manticore: Manticore is a symbolic execution tool for analyzing smart contracts that use the EVM. It can identify vulnerabilities such as integer overflow and underflow, reentrancy attacks, and more.

**NAME**: Shubham Golwal

**BRANCH**: TECOMPS

**UID**: 2020300015

References:

1. <https://blaize.tech/article-type/9-most-common-smart-contract-vulnerabilities-found-by-blaize/>
2. <https://ieeexplore.ieee.org/document/9667515>
3. <https://ethereum.org/en/>