Ethereum Smart Contracts

Devansh Patel
Information System Security
Concordia University
Montreal, Canada
pa_devan@live.concordia.ca

Darshit Gajjar
Information System Security
Concordia University
Montreal, Canada
d_gajj@live.concordia.ca

Rushi Chandalia
Information System Security
Concordia University
Montreal, Canada
ru_chan@live.concordia.ca

Kamal Sharma
Information System Security
Concordia University
Montreal, Canada
kamal.sharma@mail.concordia.ca

Palak Kaur Sodhi
Information System Security
Concordia University
Montreal, Canada
p_sod@live.concordia.ca

Sachin Verma
Information System Security
Concordia University
Montreal, Canada
v_sachi@live.concordia.ca

Gulshan Joshi
Information System Security
Concordia University
Montreal, Canada
gu_joshi@live.concordia.ca

Varsha Vivek
Information System Security
Concordia University
Montreal, Canada
va_vivek@live.concordia.ca

Abstract— To provide flash loan to users, the Unstoppable and Naive Receiver contract was created. In this paper, we thoroughly examined both contracts and found a number of weaknesses that attackers could take advantage of. We discovered that the contract is particularly susceptible to denial-of-service and reentrancy attacks. We also suggested a number of mitigation techniques, such as building fail-safe methods and providing input validation, to address these issues. Overall, our findings emphasize the significance of carrying out exhaustive security evaluations and putting in place suitable security controls to guarantee the security of DeFi applications.

Keywords—smart contracts, ethereum, blockchain, vulnerability, code analysis, exploitation.

I. INTRODUCTION

A smart contract is a self-executing commitment with integrated lines of code that define the parameters of the agreement between both the contract's counterparties. A smart agreement serves as a digital replica of the traditional paper contract that actively maintains and carries out its conditions. The smart contract is performed over a blockchain system, and the network's various computers, each contain a copy of the contract's script. This guarantees a safer and more open implementation and execution of the contract terms. Furthermore, since the code of a smart contract is checked by every member of the blockchain network, smart contracts do not need an intermediary to be validated. The cost to counterparties is diminished significantly by eliminating the intermediary from the deal. Blockchain technology serves as the foundation for the notion of smart contracts.

A blockchain is a distributed network made up of a continuously expanding list of records (blocks) connected by encryption. Unlike a traditional database, a blockchain technology does not have a single central location. Each machine on the network has permission to view the information that is recorded in the blockchain. The network is consequently less susceptible to mistakes or attacks. A data on one device cannot be changed in a blockchain without also updating the identical information on other computers in the network. With a blockchain, transactions are organised into blocks that are connected by a chain. Only once the preceding block is finished is a new block formed. Each block comprises a cryptographic hash of the preceding block and is presented in a linear chronological sequence.

There are multiple steps involved in working of smart contracts. The contract's parameters should initially be decided by the counterparties. The completed contractual requirements are then converted into a computer program. In essence, the program contains a variety of conditional statements that outline several circumstances for a potential future transaction. As a piece of code is written, it is copied across the blockchain's users and saved in the network. The code is then compiled and executed by all machines on the network. The appropriate transaction is carried out if a condition of the contract is met and has been confirmed by every user of the blockchain network.

Code analysis plays a major role in this process. Basically, code analysis investigates programming code to discover potential pitfalls or mistakes, assure compliance with coding standards, and guarantee quality. Code analysis can be done statically (viewing the program without running it) or dynamically (executing it and evaluating its performance). There are several tools and methodologies for doing code review.

II. CONTEXT

A. Common Types of Vulnerabilities in Ethereum Smart Contracts:

Smart contracts, compared to numerous other contracts, are mainly focused on monetary assets. As a result of the Blockchain's immutability, failures in smart contracts cannot be corrected after they've been implemented. Smart contract flaws might be harmful to security, as well as a tempting aim for suspicious computer hackers. In fact, regardless of whether there are outer manipulators, there is a risk of investment breakdown and economic difficulties in certain instances.

Here are some common vulnerabilities:

Reentrancy:

A risk in which an intruder can approach a contract feature recurrently before the preceding call has finished, likely to result in the intruder trying to execute their own script within the contract.

Integer overflow and underflow:

A form of security vulnerability in which a computation generates an outcome that is greater or less than the highest or lowest value that can be kept in a variable, resulting in malicious activity.

Unused variables:

Variables that are declared but not utilized can imply a coding error and may result in possible errors.

Uninitialized storage pointers:

A threat in which a memory pointer is not initialised prior to application, resulting in errors.

Function visibility:

Proclaimed public functions can be called by anybody, which includes hackers, which could also ultimately lead to security flaws.

Replay signatures attacks:

Signatures allow one account to transfer payments from another account to the blockchain. The actual account will sign a message, and the delivery account will transmit it to a smart contract so that the money transfer fees are paid by the shipping account rather than the main account.

Block gas limit vulnerability:

The block gas limit assists in preventing blocks from becoming too large. If a transfer of funds absorbs excessive gas, it's unlikely to fit in the block and will thus be ignored. Therefore, if information is maintained in arrays and then retrieved via loops over such arrays, the exchange may exhaust its gas and receive a refund. It might result in a denial-of-service (DoS) attack.

Using the Block Hash Function:

By using the block hash function, comparable to the timestamp reliance, is a strategy of attempts to hack smart contracts. It is not suggested that it be utilized for vital parts for the identical purpose as timestamp dependency: miners can modify such features and alter the funds that are withdrawn to their own benefit. This would be particularly apparent whenever the block cipher is employed as a generator of arbitrary numbers.

Incorrect Calculation of the Output Token Amount:

A large variety of actions in the contract argument are linked to token transactions to and from the contract. It opens a wide range of possibilities for errors related to calculating appropriate proportions, service charges, and revenues. That is why it is critical to work with a trustworthy token advancement business to guarantee the achievement of your endeavour.

The following are the most common errors: Incorrect digits processing, especially when interacting with a token like USDT; inaccurate command of procedure during service charge estimations, leading to substantial precision failure; and the precision incessant that was genuinely neglected in the maths processes.

B. Overview of Code Analysis

Code analysis can be done using a variety of tools and methods. They can be roughly divided into two types: static code analysis and dynamic code analysis.

Before releasing a code, static code analysis is performed to discover numerous common coding issues. It implies manually inspecting the code or using tools to automate the procedure. Analysis of static code software can examine a program without running it. They could detect syntax errors, ensure coding guidelines are adhered to, and seek out

possible data breaches. It is frequently conducted before the code is deployed, at an early stage of development. It can aid in the identification of difficulties that may be hard to miss during diagnostics and save both money and time by detecting issues prior to become quite severe.

Dynamic code analysis comprises deploying the code and observing its performance. They can detect runtime issues, test the program's efficiency, and ensure that it operates correctly. It is usually done after the program has been published. It can detect issues that were not obvious throughout assessment, such as performance problems or security flaws that arise only under certain circumstances.

C. Smart Contract Code Analysis Tools

Oyente:

Oyente is a security analytical technique for Ethereum smart contracts that is publicly available. It employs optimization technique to investigate all possible smart contract execution paths and identify potential security flaws such as integer overflows, reentrancy issues, and other flaws. Development teams can use Oyente to check the security of their smart contracts prior to deploying them to the Ethereum blockchain, ensuring that the contracts are protected and feature as destined. Security researchers use Oyente to find flaws in smart contracts that are currently utilised to the Ethereum blockchain. Oyente can spot possible security flaws as well as provide advice on how to fix them by analysing the contract's bytecode. This is especially true for smart contracts that manage large amounts of money, like the ones utilized in decentralised finance (DeFi) applications.

However, Oyente is no longer actively retained and has been primarily supplanted by other safety analysis software such as Mythril and Manticore, which offer more useful capabilities and greater services for the newest version of the Solidity computer language.

Slither:

A smart contract analyzer called Slither is free and statically evaluates the program. Solidity Abstract Syntax Tree (AST) is obtained from the source code of Solidity Compiler and is accessible to Slither. The Slither tool is made up of several analyzers that can find weaknesses in the contract. Additionally, it offers ideas for bettering the code and provides rich visual data about the contract. Slither offers a communication API. Abstract Syntax tree is a data architecture that is commonly used with compilers to portray code structure. The abstract syntax tree (AST) is a tree structure which symbolizes the source code in a technical way. So, every link in the tree is a code construct.

Slither generates a human-readable overview of the contract. It also recognises and reflects the contracts acquired by a particular contract using a diagram. Slither is published in Python, so it requires Python 3 to install and run. Analysis tools can produce false positives as well as false negatives. A false positive is when a tool claims a problem that is not essentially a mistake. False negatives are mistakes that go completely unnoticed. According to a number of similar research, several flaws may be identified in concept by static analysis techniques but are not discovered in practice due to tool restrictions.

The structure is presently utilized for the specified objectives: A wide range of smart contract errors can be discovered with no requirement for human interference or extra configuration work. Slither finds code optimization techniques that the compiler does not. Slither sums up and showcases contract details to help you in your runtime environment research. Slither can be interacted with via its API. Slither can identify various security flows such as reentrancy, function visibility, uninitialized storage pointers, unused variables and integer overflow and underflow.

Slither is a Python package that needs Python 3+ to be downloaded on the system. It pairs well with the tool solc-select, which allows users to select from multiple variations of the Solidity compiler. We can run the following lines in the terminal to install slither and solc-select.

$ pip3 install slither-analyzer

$ pip3 install solc-select

Securify:

Securify is an Ethereum smart contract security analysis tool. It is employed to detect potential security flaws in smart contracts prior to their deployment to the Ethereum blockchain. To recognise possible safety risks such as reentrancy attacks, integer overflows, and other common security flaws, the tool uses a mixture of static and dynamic analysis methods. Programmers can use Securify to check the stability of their smart contracts during the development process, ensuring that the contracts are stable and feature as intended. Security experts may additionally employ the device to identify flaws in smart contracts that are currently utilized to the Ethereum blockchain.

Securify outperforms other security analytical techniques due to its incorporation with Remix, a famous web-based application framework for Ethereum smart contracts, interoperability with the newest version of the Solidity programming language, and advanced analytical skills.

Mythril:

Mythril is an open-source vulnerability assessment tool for analyzing smart contracts published in Solidity, the Ethereum blockchain's most widely used programming dialect for smart contracts. It is intended to identify possible security flaws in smart contracts and make proposals to enhance the code. Mythril operates through symbolic execution, a method for analyzing a program's behavior by discovering all available options it can take. Mythril examines the bytecode of the smart contract and creates a control flow diagram that depicts all different possibilities that the code can accept.

Mythril can identify various vulnerabilities like Reentrancy, Integer overflow and underflow, Out of gas which is a threat in which a contract consumes more gas than what is available, resulting in the transfer of funds failing and the last one is solidity compile bug which is a risk caused by a virus in the Solidity compiler that can result in unforeseen actions.

Setup of Mythril on various systems:

PyPI on Mac OS:

brew update

brew upgrade

brew tap ethereum/ethereum

brew install solidity

pip3 install mythril

PyPI on Ubuntu:

Update

sudo apt update

Install solc

sudo apt install software-properties-common

sudo add-repository ppa:ethereum/ethereum

sudo apt install solc

Install libssl-dev, python3-dev, and python3-pip

sudo apt install libssl-dev python3-dev python3-pip

Install mythril

pip3 install mythril

myth version

Manticore:

Manticore is a symbolic execution tool that analyzes smart contracts and binary files. It is primarily used mostly for Ethereum smart contract security assessment, allows programmers and security experts to recognise security flaws in the code and enhance the contract's security level. Manticore utilizes symbolic execution to investigate all plausible smart contract execution paths and recognise possible safety risks such as integer overflows, reentrancy vulnerabilities, and other popular smart contract security flaws. Manticore is suitable for performing tasks apart from security analysis, such as bug identification, test case generation, and contract validation. Manticore may be employed as a stand-alone tool or merged into other tools and procedures like the Truffle framework, Remix IDE, and other Ethereum design tools. This tends to make it a flexible and potent tool for optimizing the safety of smart contracts.

All in all, Manticore is a strong and adaptable tool for analyzing Ethereum smart contracts. It employs symbolic execution to investigate all possible execution paths of a contract and identify potential security vulnerabilities, making it an indispensable tool for developers and security researchers seeking to ensure the security and reliability of their smart contracts.

Manticore's procedure can be categorized as follows:

Manticore analyzes the bytecode or source code of a smart contract as input to identify the various functions and variables in the contract. Manticore employs symbolic execution to investigate all potential available options of the smart contract, beginning with the contract's initial state. Manticore creates a collection of constraints that reflect the circumstances needed for executing each route as it examines various operation pathways. Manticore then employs a constraint solver to ascertain whether every collection of limitations is acceptable, — in other words, how an input collection remains that fulfils the restrictions. If Manticore discovers a collection of requirements that is satisfactory, it indicates that the contract contains a possible vulnerability that an intruder could manipulate. Eventually, Manticore produces an analysis that enumerates all of the possible risks found in the smart contract, as well as suggestions regarding how to solve them.