

Project Group Members:



Name: Rehan Mumtaz

Roll No: TESE-19036

Email: mumtaz4203271@cloud.neduet.edu.pk



Name: Shiekh Muhammad Ahsan Tariq

Roll No: TESE-19054

Email: tariq4204652@cloud.neduet.edu.pk



Name: Kabeer Ahmed

Roll No: TESE-19028

Email: ahmed4201750@cloud.neduet.edu.pk

Smart Security in Smart Contract

Rehan Mumtaz ¹, Shiekh Muhammad Ahsan Tariq ², Kabeer Ahmed ³

^{1,2,3} Department of Software Engineering, NED University of Engineering & Technology, Karachi, Pakistan

Abstract-Smart contracts are contracts for any kind of agreement, but what makes them smart is that they run as code automatically on a blockchain. They implement the blockchain's decentralization idea by automating transactions in the absence of any centralized control. Although smart contracts have advanced quickly, there are still a number of security issues that need to be explored. Loss in money is frequently caused by such weaknesses. In the present world, a fully automated solution to vulnerability detection is desired. We summarize the security challenges of smart contracts. It is merely a human-made, error-prone computer application. The majority of smart contract flaws are caused by security issues in lines of computer code. Future scholars will receive full analyses and recommendations from our prestigious work. As we proposed methods of securing smart-contract through different approaches by taking in place of the secure coding practices, we analyze the success rate is much higher than relying on the tool. Our aggressive analysis of vulnerability in a smart contract depicted that complex logic sometimes provides a foothold to the attacker, the main point is to remediate it before anyone can exploit it or increase the severity. We examine the methods and tools that developers can use to safeguard data against hacking.

Keywords— smart contract; Ethereum; SmartCheck, security concerns; vulnerabilities; Xpath

I. Introduction

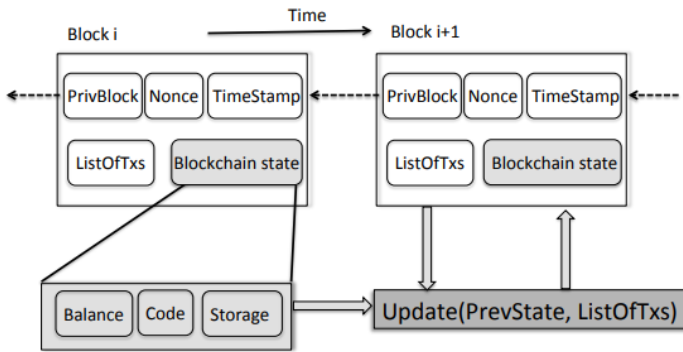
Blockchain was originally described in 1991 as

was used for securing original digital documents. This technology remained unused for many years until in 2009 Satoshi Nakamoto and the first ever digital cryptocurrency was developed called Bitcoin. The blockchain technology has a property that once some data is recorded in them, it is difficult to alter them. The unique hash of every block is stored in the next block. The hash is created at the time of storing of data and if the data is changed its hash is also

changed. All blocks contain the hash of previous blocks. So, if any block is tempered by all other blocks (nodes) in the network. Instead of a central server, Block chain uses peer-to-peer connection.

This technology is constantly evolving. One of its developments is the smart contracts. It means the code that runs on blockchain and it ensures an agreement between two parties. They are like normal contracts except they are digital and are programmed in a special programming language called solidity. We don't need to rely on third parties. It's just you and the person who is receiving your transactions within the decentralized networks of blockchain. Smart contracts can hold all the received funds until a

certain goal is achieved. If the goal is achieved the money is transferred to the creator, if it fails then it is sent back to senders. But why should we trust smart contracts? Because it is immutable and distributed, means that it is not possible to alter the smart contract. Secondly, it is validated by everyone in the network. A part of crowdfunding, the smart contracts are used in insurance claims, postal to deliver payments, electronic voting systems etc. Nowadays there are a number of block chains that support smart contracts. The most popular among them is the Ethereum block chain. It is developed for creating smart contracts.



The design in famous cryptocurrencies like Ethereum. Every block is made up of several transactions.

But in spite of a number of security measures there is also a risk of detrimental attacks. A wide range of attacks are made on blockchain technologies related to cryptocurrency, E-wallets, transactions, smart contracts etc. Some famous attacks on smart contracts are DAO attack and King of Ether Throne. So, the security factor should be properly focused to avoid any kind of vulnerabilities.

There are some tools available such as Oyente, Osiris, Remix. In this paper we discuss a security tool called SmartCheck. Its performance is more accurate as compared to the others. It is a tool for checking the security in smart contracts. It does not only detect the vulnerabilities but also identifies the cause of

vulnerabilities along with details. It also provides some recommendations related to particular security issues. It uses XPath queries to detect the patterns of vulnerabilities. An experiment was held to check the efficiency of SmartCheck by providing around 4.5k contracts and it was successful in detecting vulnerabilities in 99.9%.

Following are the descriptions of sections in this research paper. In Section II the literature review of some research papers is discussed. In Section III tools, section IV proposed architecture ,section V the methodology to implement smart security in smart contracts is described. The Section VI comparative analysis, VII conclusion & VIII contains bibliography..

II.RELATED WORK

This section involves reviews concentrated on smart contracts, its security and attacks which helps in investigating the security and vulnerabilities in smart contracts. Some reviews of related work are described as under:

The paper [1] presents some mistakes that are made in smart contracts and guide the people with some methods to prevent these mistakes. In this paper the students are targeted which are involved in the practices of smart contracts. The common mistakes include some programming, logical and security issues. The programming and logical errors commonly occurred in the encoding of complex state machines. The most common pitfall that occurs in smart contracts is the error that causes monetary leaks.

Smart contracts developed on blockchain have a huge impact on new businesses. It is a difficult task and started only in industrial and scientific fields. Wohrer et al. [2] presented security patterns with solutions based on grounded theory, applied on smart contract

data. These solutions are based on Solidity which is a popular programming language used in block chain. These security patterns can be applied to solve the security issues to avoid common attack scenarios.

Some of the vulnerabilities along with attacks in the architecture layer of smart contracts are discussed in paper [3]. Three aspects which are vulnerabilities, attacks and defense are presented in this paper. It presents a survey of Ethereum security systems. It presents 40 types of vulnerabilities in the architecture of Ethereum and also provides its root causes. Some defense techniques are discussed. It also highlights a factor that using a better programming language can help in making contract fault tolerant but cannot make the contract secure.

A new way for the transactions of cryptocurrency is smart contracts but there are a lot of vulnerabilities in this technique. An exploitable bug can cause a loss of a huge amount of money. A most common type of bug known as reentrancy bug is discussed in paper. This bug caused an attack known as DAO (Decentralized Autonomous Organization) in 2016, which resulted in a debt of \$60 million worth of Ethereum cryptocurrency. Liu [4] presents a technique called *ReGuard* for the detection of bugs. It is a fuzzing-based technology that performs fuzz testing by generating random transactions. It identifies vulnerabilities and automatically flags them.

A framework known as Osiris, is presented in research [5]. This technique is designed to find bugs and errors in smart contracts. This tool works efficiently and detects a wider range of errors than the tools available. An experiment was performed to evaluate its performance, a large dataset is provided that contains around 1.2 million smart contracts. Smart contracts are being used in business and dealing with intellectual properties. But due to certain

vulnerabilities in smart contracts many issues are reported which causes great financial losses. Many solutions have also been developed. After analyzing many researches Y Haung [6] addresses some vulnerabilities and like traditional software development lifecycle (SDLC) the contract lifecycle are also divided into four stages.

Checking the mechanism of transaction, seemed to require much attention as it is a critical function in smart contracts when it is dealing with live transfer of money. This makes it critical as to secure this requires different techniques. Some go for manual while some for automated prevention technique, while in [7] Jemin Andrew came up with idea of run time validation that found to be a strategic and efficient approach. However there is a downfall for this method which is overhead of runtime validation (excessive performance overhead), which can be costly for domains in blockchain. This issue can be integrated with Solythesis, tool of runtime validation for smart contracts (Ethereum), it uses as source compiler tool for solidity and analyzing/detect of error at compile time, however it also facilitates expressive or mechanism language (which uses quantifiers for allowing users to identify important functions of smart contracts)

In a research study, a method posted in [9] about checking of smart contract if it exhibits vulnerability by using model checking and adopting rules of μ -Calculus taking care of vulnerabilities normally termed as Automata. It was thus evaluated on a dataset and predicting it through a precision and recall rate whether it is vulnerable or not. While other researchers gave their approach, one of the most maximax mechanisms that found to be useful is analyzing access control mechanisms in blockchain as it can lead to deadliest bugs that can be exposed to a greater harm for an entity. In [8] the paper, researcher think of this idea of using High

Granularity Metrics scheme, building white and blacklisting access control in an adaptive manner to filter out unwanted and unauthorized pickling with data. Thus we create many layers for learning, identifying, and alerting unusual behavior by increasing visibility through aggregated feature-level measurements. Another report by researchers shows in their paper is [10], the introduction of tool Ethainter, a security analytics tool that cleans data from smart contracts to verify the information flow. Ethainter identifies fusion attacks that result in significant breaches by spreading contaminated information across numerous transactions. The analysis covers the entire blockchain, which consists of millions of accounts and hundreds of thousands of different smart contracts. We validate that Ethainter is more accurate than earlier methods for autonomous mining generation.

Researchers nowadays always look for a way to secure the transactions carried out in blockchain due to the critical nature. False positive is an issue usually addressed by many to use methods to overcome it but still it persists which If the problem is either not present or cannot be exploited. In one of the studies [11], a formal approach part work is presented to address that issue. Our work builds upon our earlier analysis consisting of two stages: first, by considering the notion. Focus on the switch's function calls and LTL attributes to find the accuracy of the smart contract. These characteristics may be unique to the management or information flow of the contracts under verification. By suggesting formal characteristics of LTL for vulnerabilities from the literature, we show that they can also be utilized to reveal flaws. In another study, [12] another tool HFContract-Fuzzer is introduced, a Fuzzing-based approach that combines a go-fuzz-based Fuzzing engine with go-written smart contracts. We identified four of the five agreement vulnerabilities we found

from well-known reassets using HFContractFuzzer, proving the efficacy of the suggested approach.

In the paper [13], authors present a number of fresh security issues where a malicious party might influence the execution of a smart contract in order to profit. These flaws point to minute weaknesses in the platform's underlying distributed semantics knowledge. They suggest approaches to improve Ethereum's operational semantics in order to make contracts more secure. They created a symbolic execution tool called Oyente to help contract writers for the current Ethereum system detect possible security flaws. Oyente classifies 8, 833 of the 19, 366 currently in use Ethereum contracts as insecure, including the The DAO flaw that cost the company \$60 million in June 2016. A number of studies with published code that are used in attacks made on Ethereum network are also taken into consideration.

Researches showed security flaws, faults, and vulnerabilities in the Ethereum smart contract. Applying the Self Destruct feature is the most effective way to terminate a contract at the blockchain device and transfer all of the Ethers included within the contract stability. As a result, when issues are discovered, many builders use this feature to cancel the agreement and set up a new one. A deep learning-based algorithm was used in one study [14] to retrieve the updated version of a destroyed contract in order to identify security flaws in Ethereum smart contracts. Then, using open card sorting, we look for security flaws in the upgraded versions. In another study [15] Authors personally found their defined contract flaws in 587 actual smart contracts by examining Feedback; they then made their dataset available to the public. Finally, they listed five effects brought on by contract flaws. These aid developers in comprehending the problems' symptoms and order of importance for elimination. Another study's say [16] The authors explore the

topic of security of intelligent settlement programming in addition to providing an extensive taxonomy of all accepted security concerns and examining the safety code evaluation techniques utilised to identify acknowledged vulnerabilities. Through comparing their effectiveness and precision on acknowledged faults on a consultant pattern of weak contracts, they investigate Ethereum's safety code evaluation tools. To determine how comprehensive the most recent safety evaluation tools for Ethereum are, researchers looked at the performance of 4 safety tools—Oyente, Securify, Remix, and SmartCheck—as well as 21 stable and 24 tilted contracts.

The author gives [17] Their research sheds light on the smart-contract security field. Based on thorough research and excellent experiments we present an updated taxonomy of vulnerabilities, their architectural categorization, along with their severity level. They also examine some of the security technology to assess its accuracy, strength, and consistency. They have categorized the equipment in accordance with the manner their user interface and capacity for evaluation enable them to provide a "kingdom of the art" set of protection equipment on Ethereum. They create a matrix of protective gear and the vulnerabilities it covers in order to find gaps and missing vulnerability checks. In another work [18], they provide a security assurance approach for smart contracts in order to prevent potential vulnerabilities while creating smart contracts. Their solution has two essential components. Developers can benefit from clearer understanding of their code structure with the topology diagram creation of invocation relationships for cross-file smart contracts. In addition, they broaden the range of logical dangers and may identify and pinpoint them using syntax analysis and symbolic execution. Together with syntax analysis, they can identify logic vulnerabilities to certain functions. They created the

SASC tool based on these characteristics. It can offer Ethereum smart contracts with very strong quality assurance.

III. TOOLS AND TECHNOLOGIES

Smart contracts: Unsurprisingly, a smart contract is a set of files executed on the blockchain that is linked to a specific environment. These are frequently employed to automate the merger execution process. This is because, barring the involvement of certain middlemen and the absence of events, all stockholders can now be confident in the outcome. While the environments are connected, workflows can be automated and additional procedures can be started.

ERC-20 Token: Numerous cryptocurrencies have been developed in the modern era. However, the majority of them are accomplished via Ethereum-based smart contracts, often referred to as tokens, which also hold their own blockchain configuration. Ethereum offers a variety of memorial flags that reflect the opinions of related coins. In this scenario, numerous tokens can interact properly and be reused by diverse users (like, wallets and exchange markets).

Remix: Users of all skill levels use the Remix IDE during the full process of developing a smart contract. It doesn't require any setup, promotes a quick development cycle, and comes with a great selection of plugins and a straightforward GUI. The IDE is offered as a desktop package, an online application, a VS Code extension, and other formats.

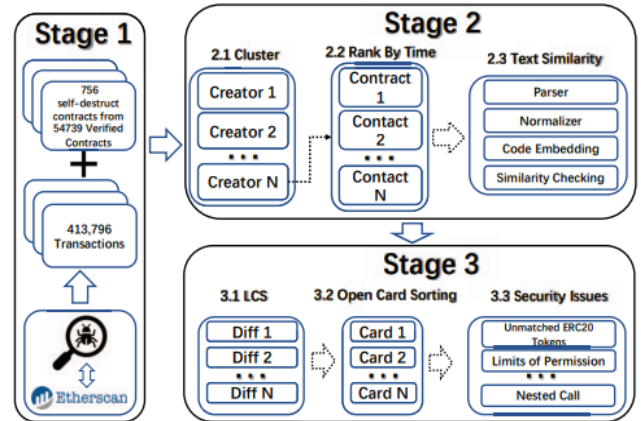
SmartCheck: The result of a thorough investigation to identify more code explosions and problems in Ethereum Smart Contracts registered under the Solidity installation term is known as SmartCheck.

Securify: On their website, Securify describes itself as a net-positioned security platform with industrialized components (to enable everyone to check smart contracts), assurance (for precise risk), and flexibility. I promise (for some of these). days to seize a particular exposure). Securify uses unambiguous evidence but also relies on tests of inflexible reasoning. Project reordering, repeated calls, insecure codified styles, unexpected heavenly floods, and the usage of inaccurate recommendations are the main difficulties it addresses. Repeated calls, unexpected downpours of rain, and one of the related codified style checks are warranted, nevertheless (sufficient approach requirement).

F* Framework: Microsoft Research's F* provides a framework for evaluating runtime security and functional correctness of Ethereum smart contracts. By converting Solidity or bytecode to F* (Functional Programming Language), use formal verification to uncover potential vulnerabilities such as re-entry and exception errors.

Oyente: Oyente uses symbolic execution to identify possible security flaws, such as reentrancy, timestamp reliance, mishandled exceptions, and dependence on transaction sequencing. The tool has the ability to examine both Solidity and a smart contract's bytecode. It could only be used via a command line interface in its early stages. It currently offers a web-based interface that is more user-friendly.

Architecture overview for comparing older versions to identify security flaws in Ethereum smart contracts.



B. Smart Contract Design:

Anyone can use a smart contract's functionality after it has been installed on the Ethereum network. The feature can have security measures that prevent users from utilizing it.



Smart Contract

Ethereum Account Type (Just like User Account)



Address — 0x16E0022b17B...
 0 Ether
 Balance — contract Counter {
 uint counter;
 }
 Code — function Counter() public {
 counter = 0;
 }
 State — function count() public {
 counter = counter + 1;
 }
 }

IV. PROPOSED ARCHITECTURE

A. Architecture Design:

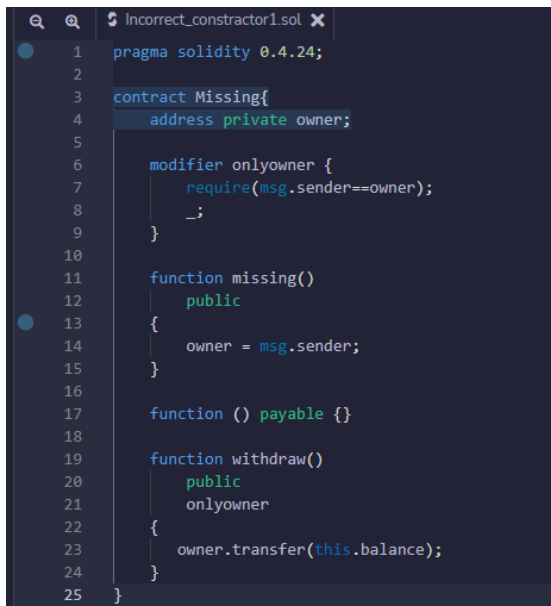
V. METHODOLOGIES

A) Incorrect Constructor Name Vulnerability

An unusual element(*constructor*) known as a function `Object() { [native code] }` is useful just when understanding creation is taking place. They routinely carry out advantageous developments that are important, like Assurance of Contract Owner. Prior to Solidity version 0.4.22, creating a component with the same call as the constructor's understanding polish was the most practical approach to create a function `Object() { [native code] }`. If a component's call no longer exactly matches the understanding call, it becomes a regular callable element and is no longer considered to be a function `Object() { [native code] }`. This can lead to security problems, especially if you repeat sneaky understanding code inside a restricted call and neglect to rename the function `Object() { [native code] }` highlight in tandem.

Incorrect_constructor1.sol


In this file on line 11 we use function `missing()` as a constructor which is the incorrect way for declaring the constructor.



```
1 pragma solidity 0.4.24;
2
3 contract Missing{
4     address private owner;
5
6     modifier onlyowner {
7         require(msg.sender==owner);
8         _;
9     }
10
11     function missing()
12     public
13     {
14         owner = msg.sender;
15     }
16
17     function () payable {}
18
19     function withdraw()
20     public
21     onlyowner
22     {
23         owner.transfer(this.balance);
24     }
25 }
```

Incorrect_constructor2.sol

In this file on line 11 we use function `Constructor()` as a constructor which is also the incorrect way for declaring the constructor.



```
1 pragma solidity 0.4.24;
2
3 contract Missing{
4     address private owner;
5
6     modifier onlyowner {
7         require(msg.sender==owner);
8         _;
9     }
10
11     function Constructor()
12     public
13     {
14         owner = msg.sender;
15     }
16
17     function () payable {}
18
19     function withdraw()
20     public
21     onlyowner
22     {
23         owner.transfer(this.balance);
24     }
25 }
26
27 }
```

Prevention

The function `Object() { [native code] }` definitions are made clearer with the introduction of a brand-new function `Object() { [native code] }` key-word in Solidity model 0.4.22. Therefore, it is advised to enhance the agreement to a recent version of the Solidity compiler and extradate to the brand-new function `Object() { [native code] }` declaration.

Correct_constructor.sol

In this file on line 11 we use `Constructor()` as a constructor which is the correct way for declaring the constructor.

Contracts that take delivery of statistics and use them in a sub-name on any other contract are subject to inadequate gas griefing attacks. Both the entire transaction is reverted and execution is continued if the sub-name fails. When employing a relay contract, the person who conducts the transaction, known as the "forwarder," can successfully censor transactions by using just enough gas to complete the transaction but insufficient gas to ensure the success of the sub-name.

Relayer.sol

In this file on line 17 function relay() uses just enough gas to submit the transaction, however insufficient subcall to process

```
1 pragma solidity ^0.5.0;
2
3 contract Relayer {
4     uint transactionId;
5
6     struct Tx {
7         bytes data;
8         bool executed;
9     }
10    mapping (uint => Tx) transactions;
11
12    function relay(Target target, bytes memory _data) public returns(bool) {
13        require(transactions[transactionId].executed == false, 'same transaction twice');
14        transactions[transactionId].data = _data;
15        transactions[transactionId].executed = true;
16        transactionId += 1;
17        (bool success, ) = address(target).call(abi.encodeWithSignature("execute(bytes)", _data));
18        return success;
19    }
20 }
21
22 contract Target {
23     function execute(bytes memory _data) public {
24     }
```

Prevention

You have two options to prevent inadequate gas grieving: either only allow transactions to be relayed by customers, or demand that the forwarder provide enough gas..

Solved_relayer.sol

In this file on line 11 we pass the parameter `_gasLimit` to the function relay() to to use the required amount of gas and also take `_gasLimit` as an input from the user, the required amount of gas for transaction by showing the gas amount left for them on line 19-21.

```
1 pragma solidity ^0.5.0;
2
3 contract Relayer {
4     uint transactionId;
5
6     struct Tx {
7         bytes data;
8         bool executed;
9     }
10    mapping (uint => Tx) transactions;
11    function relay(Target target, bytes memory _data, uint _gasLimit) public {
12        require(transactions[transactionId].executed == false, 'same transaction twice');
13        transactions[transactionId].data = _data;
14        transactions[transactionId].executed = true;
15        transactionId += 1;
16        address(target).call(abi.encodeWithSignature("execute(bytes)", _data, _gasLimit));
17    }
18 }
19
20 contract Target {
21     function execute(bytes memory _data, uint _gasLimit) public {
22         require(gasleft() >= _gasLimit, 'not enough gas');
23     }
24 }
```

D)Reentrancy:

Vulnerability

The “Reentrancy” means the function of a contract is called by an external contract which contains malicious code as the malicious code reenters the contract. This will invoke the function to behave in undesirable ways.

In the reentrancy attack, a malicious contract calls back into the calling contract before the first invocation of the function is finished. One of the major dangers of calling external contracts is that they can take over the control flow. It is carried out by an attacker by creating a contract. This contract has a malicious code in its “receive” function when anyone sends ether through this contract. This malicious code will perform the functions that the code is not intended to.

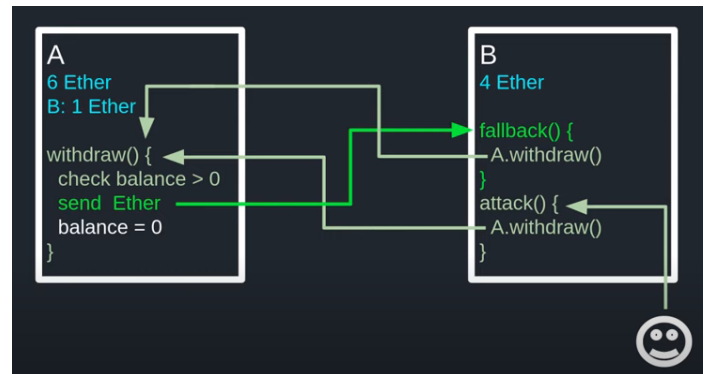


Fig. Mechanism of Re-entrancy attack

Let us consider the code of a vulnerable contract. This contract which allows a person to take out 1 ether in a week

File Name: *EtherStore.sol*

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.6.0;
3 contract EtherStore {
4     uint256 public withdrawLimit = 1 ether;
5     mapping(address => uint256) public lastWithdrawTime;
6     mapping(address => uint256) public balances;
7     function depositFunds() public payable {
8         balances[msg.sender] += msg.value;
9         (bool sent, ) = msg.sender.call{value: 1 ether}("");
10        require(sent, "Failed to send Ether");
11    }
12    function withdrawFunds (uint256 _weiToWithdraw) public {
13        require(balances[msg.sender] >= _weiToWithdraw);
14        require(_weiToWithdraw <= withdrawLimit);
15        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
16        (bool success,) = msg.sender.call{value: balances[msg.sender]}("");
17        require(success, "receiver rejected ETH transfer");
18        balances[msg.sender] -= _weiToWithdraw;
19        lastWithdrawTime[msg.sender] = now;
20    }
21 }

```

Fig. EtherStore.sol contract code

Here two functions are declared; 1) depositFunds and 2) withdrawFunds.

a) depositFunds Function:

This function is designed to increase the balance of the sender.

b) withdrawFunds Function:

This function is designed to set the amount of wei to be taken out by a sender.

This function will be executed only if the sender has not made any withdrawal in the week. Secondly, the amount of ether that is to be withdrawn must be less than one ether.

In the above contract the vulnerability is present in line 16 at the point when the amount of ether requested by the user is sent by the contract.

Consider a malicious code that is developed by an attacker:

File Name: Attacker.sol

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.5.0 <0.7.0;
3 import "EtherStore.sol";
4 contract Attack {
5     EtherStore public etherStore;
6     constructor(address _etherStoreAddress) public {
7         etherStore = EtherStore(_etherStoreAddress);
8     }
9     function pwnEtherStore() public payable {
10        require(msg.value >= 1 ether);
11        etherStore.depositFunds{value: 1 ether}();
12        etherStore.withdrawFunds(1 ether);
13    }
14    function collectEther() public {
15        msg.sender.transfer(address(this).balance);
16    }
17    // fallback function - the actual malicious code
18    receive() external payable {
19        if (address(etherStore).balance > 1 ether) {
20            etherStore.withdrawFunds(1 ether);
21        }
22    }
23 }

```

Fig. Attack.sol contract containing the malicious code

Working of code:

The malicious contract is first developed by an attacker at the address given by the contract written on the EtherStore.sol file. This contract will initialize a variable named as etherStore present on the vulnerable contract. Then the attacker will use the attackEtherStore function and request some amount of ether. For simplicity consider the requested amount is one ether and the total balance in this contract is 10 ether. Then the code will be executed in the following way:

1. The depositFunds function of the contract present on EtherStore will be executed. The value of msg.value will be set to one ether. The malicious contract will be executed and the value of balances will be set to one ether.
2. The withdrawFunds function written in the contract of the EtherStore will be called by malicious code and the value of parameter will be taken as one. As no previous withdrawal is made so the code will be executed without any error.
3. One ether will be sent to malicious code

4. The fallback (receive) function will be executed as soon as the payment is received by malicious contract

5. The remaining balance of the EtherStore contract will be 9 on the execution of the if statement.

6. withdrawFunds will be called again from EtherStore. Hence it makes reentrancy to EtherStore contract

7. When the withdrawFunds function is called again, the line 17 of the EtherStore.sol file will not be executed before calling the function as it is written inside the function body so it will run on executing the function. Thus, the attacker's balance will be one. This is also the passing criteria of the lastWithdrawTime variable. So the function will be executed again.

8. The contract written by the attacker will get one ether again.

9. The attacker code will then remain running every time until the condition `EtherStore.balance > 1` written in Attacker.sol file is met.

10. The execution will be stopped only when line 17 of the EtherStore contract is executed. It will be executed when only one ether or less than one ether is left in the EtherStore contract file.

11. The balance and withdrawTime variable will be mapped together and the execution will be stopped.

Finally the attacker will be succeed in taking out the full amount of the EtherStore contract but in one by one execution

Prevention:

The prevention of the vulnerabilities of smart contract is made in a number of ways:

The first technique is to use a built-in transfer function. This function will send only 2300 gas. This gas will be insufficient to call any other external contract.

Another technique is that the state variables should be changed before sending the ether. In the above example line number 17 and 18 should be placed before line number 16, so that these lines will be executed and the withdrawFunds function will be stopped on meeting the certain condition.

One more technique is that we can use mutex which means to add a state variable when the code is running. This will save the code from reentrance. This is also called as Checks-effects-interactions pattern

So, by applying these preventive techniques to the EtherStore contract the contract will be saved from reentrance. The modified EtherStore code will be:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.5.0 <0.7.0;
3 contract EtherStore {
4     // initialize the mutex
5     bool reEntrancyMutex = false;
6     uint256 public withdrawLimit = 1 ether;
7     mapping(address => uint256) public lastWithdrawTime;
8     mapping(address => uint256) public balances;
9     function depositFunds() external payable {
10         balances[msg.sender] += msg.value;
11     }
12     function withdrawFunds (uint256 _weiToWithdraw) public {
13         require(!reEntrancyMutex);
14         require(balances[msg.sender] >= _weiToWithdraw);
15         // limit the withdrawal
16         require(_weiToWithdraw <= withdrawLimit);
17         // limit the time allowed to withdraw
18         require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
19         balances[msg.sender] -= _weiToWithdraw;
20         lastWithdrawTime[msg.sender] = now;
21         // set the reEntrancy mutex before the external call
22         reEntrancyMutex = true;
23         msg.sender.transfer(_weiToWithdraw);
24         // release the mutex after the external call
25         reEntrancyMutex = false;
26     }
27 }
```

Fig. EtherStore.sol updated code

Output:

```
call to EtherStore.withdrawallimit
call [call] from: 0x58380a6a781c568545dcfc803fc8875f56beddC4 to: EtherStore.withdrawallimit() data: 0x7dd...fe78d
from 0x58380a6a781c568545dcfc803fc8875f56beddC4 ⓘ
to EtherStore.withdrawallimit() 0xf8e81d47283a594245e36c48e151709f8c19f8e8 ⓘ
execution cost 23413 gas (Cost only applies when called by a contract) ⓘ
input 0x7dd...fe78d ⓘ
decoded input {} ⓘ
decoded output {
  "0": "uint256: 1000000000000000000"
} ⓘ
logs [] ⓘ ⓘ
```

Fig. output of EtherStore.sol

E)Arithmetic Over/Underflows:

The Ethereum Virtual Machine displays fixed-size data types for entire values. This suggests that a number variable be capable of handling a particular range of values. A uint8 can store numbers between [0,255], for example. If you try to fit 256 into an uint8, the result is 0. Factors in Solidity can be abused if caution is not exercised. It is irrational to expect client input, and calculations are performed that produce results that are outside the range of the data type that is used to record them.

Vulnerability

An over/undercurrent happens when a job requires a fixed-size variable to hold a number (or other piece of information) that is outside the restrictions of the variable's information type.

As an illustration, the result of subtracting 1 from a non-negative uint8 (unsigned 8 copies) variable with a value of 0 is 255. There is a flow like that. The result is flattened and delivers the largest number that uint8 can hold as a result of the distribution of numbers below its storage capacity. Like adding 28 = 256 to uint8, folding the entire length of uint has no effect on the variable. Examples of this behavior include erroneous calculation thresholds (exceeding 2 will leave the amount unchanged) and using the vehicle's odometer to calculate kilometers (reset to

000000 above the maximum of 999999). the future). When more significant integers are added than the data type can handle, flooding occurs. Simply simply, an uint8 with a current value of 0 gets the number 1 by adding 257 to it. Fixed-size factors can occasionally be thought of as cycles. By adding the greatest number possible, you begin counting from zero in this situation. When subtracting from zero, it is stored and begins by counting backwards from the highest value. When the maximum negative value is reached, it will restart since a stamped int that can withstand negative values will tolerate it. Trying to remove 1 from an int8 with a value of -128, for instance, results in 127. Attackers can take advantage of programming and provide unexpected views by using this type of numeric trap. Think about the TimeLock agreement in TimeLock.sol. Think about the TimeLock agreement in TimeLock.sol. Such mathematical traps give an attacker the chance to take advantage of programming and produce unanticipated views. Take the TimeLock contract from TimeLock.sol as an example.

```
contract TimeLock {
    mapping(address => uint) public balances;
    mapping(address => uint) public lockTime;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] = now + 1 weeks;
    }

    function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }

    function withdraw() public {
        require(balances[msg.sender] > 0, "not enough funds");
        require(now > lockTime[msg.sender], "Lock time not expired");
        uint amount = balances[msg.sender];
        balances[msg.sender] = 0;

        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Failed to send Ether");
    }
}
```

Fig. Timelock.sol

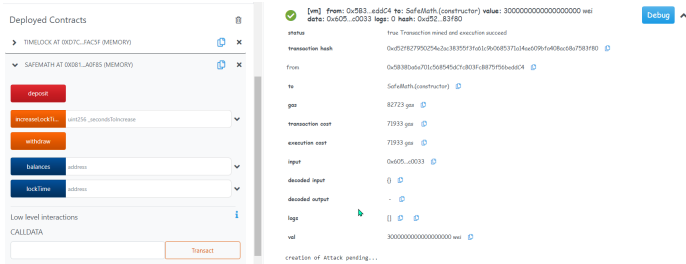


Fig. Funds Deposited

Clients can store ether into the agreement, and it will remain there for around seven days. This agreement is designed to function like a time vault. According to the terms of this agreement, the customer may extend the standby duration to more than a week if they so choose, but when kept, the client may be confident that their ether is held securely for around seven days.

It may be possible to guarantee that a client's security of their transactions temporary if they are required to submit their secret key, like in the case of an agreement. However, like if client had secured in 100 ether under this contract and turned over their keys to an aggressor, No matter the lockTime, the attacker might use a flood to get the ether.

```
contract Attack {
    TimeLock timeLock ;

    constructor(TimeLock _timeLock) public {
        timeLock = TimeLock(_timeLock);
    }

    fallback() external payable {}

    function attack() public payable {
        timeLock.deposit{value: msg.value}();
        /*
        if t = current lock time then we need to find x such that
        x + t = 2**256 = 0
        so x = -t
        2**256 = type(uint).max + 1
        so x = type(uint).max + 1 - t
        */
        timeLock.increaseLockTime(
            type(uint).max + 1 - timeLock.lockTime(address(this))
        );
        timeLock.withdraw();
    }
}
```

Fig. Attacker code

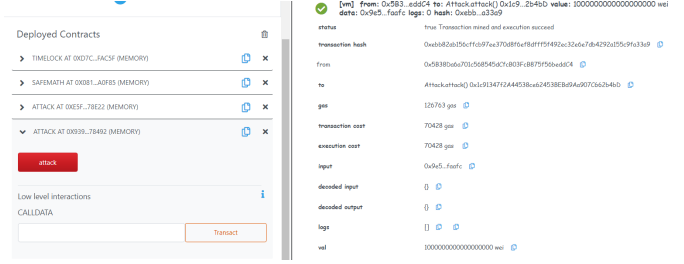


Fig. Funds Withdrawal Bypassing protection

The ongoing lockTime for the place for which they now have the key could be chosen by the aggressor (it is a public variable). This is what we should label userLockTime. Then, they might call the increaseLockTime capability and pass the value 256 - userLockTime as an argument. If this value were added to the dynamic userLockTime, a flood would follow, setting lockTime[msg.sender] to 0. The attacker might then essentially call for the ability to withdraw in order to secure their reward.

Prevention

Utilizing or creating numerical libraries that replace the standard number-related administrators expansion, deduction, and increase is the ongoing, routine strategy to prepare for under/flood weaknesses

For the Ethereum community, OpenZeppelin has built and reviewed secure libraries successfully. Its SafeMath library, prevention can be made to overflow vulnerability

Let's use the SafeMath library to correct the TimeLock contract as an example of how Solidity makes use of these libraries. The following is the agreement's flood-free version:


```

library SafeMath {
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
        return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
}
function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold
    return c;
}
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
}
function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
}
}

```

Fig. SafeMath Library Code

```

using SafeMath for uint;

mapping(address => uint) public balances;
mapping(address => uint) public lockTime;

function deposit() public payable {
    balances[msg.sender] += msg.value;
    lockTime[msg.sender] = now + 1 weeks;
}

function increaseLockTime(uint _secondsToIncrease) public {
    lockTime[msg.sender] = lockTime[msg.sender].add(_secondsToIncrease);
}

```

Fig. Some remediation in old code

```

[vm] from:0x51a...d7de9 to:Attack.attack() 0xfd7...fb116
value:1000000000000000000 wei data:0x9e5...faafc logs:0
hash:0x44f...e932c

transact to Attack.attack errored: VM error: revert.
revert The transaction has been reverted to the initial state.
Reason provided by the contract: "SafeMath: addition overflow". Debug the
transaction to get more information.

```

Fig. No Funds withdrawal

You'll see that the SafeMath library's tasks have replaced all conventional numerical activities. There is no longer any activity under the TimeLock contract that is equipped for under/flood.

VI.COMPARATIVE ANALYSIS

Vulnerability:

Analysis between different vulnerabilities showed as in the figure, which shows the impact, severity score and distribution growing percentage found normally in codebases in smart-contracts

Contract Defect	Distribution	Score	#Defects	Impacts
Incorrect Constructor Name		4.50	25 (4.26%)	IP3
Insufficient Gas Griefing		4.28	5 (0.85%)	IP2
Message call with hardcoded gas amount		4.54	5 (0.85%)	IP1
Reentrancy		4.10	84 (14.31%)	IP3
Arithmetic Over/Underflows		4.45	13 (2.21%)	IP2

Fig. Analysis of vulnerabilities

Smart Contracts:

Occurrence of vulnerabilities between unique and normal contracts as depicted in the figure

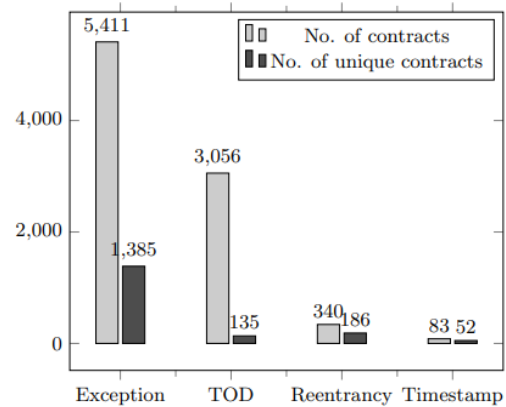


Fig. Number of buggy contracts per each security problem

Security Tools:

We compiled a list of known vulnerabilities based on a literature search and internet resources. We also followed any relevant web-articles, blogs, and

Reddit communities that discussed smart contract security concerns. Additionally, we have spoken with developers or users for a specific security product via email and group chats (Slack channels,

Gitter). We decided to use Remix which is more secure than the other tools.

Security Tool	ReEntrancy	Timestamp dependency	TOD ²⁶	Mishandled exceptions	Immutable Bugs	tx.origin usage	Gas costly patterns	Blockhash usage
Oyente	✓	✓	✓	✓	✓	X	X	X
Remix	✓	✓	X	✓	X	✓	✓	✓
F*	✓	X	X	✓	X	X	X	X
Gasper	X	X	X	X	X	X	✓	X
Securify	✓	X	✓	✓	X	✓	X	X
S. Analysis	X	X	X	✓	X	X	X	X
SmartCheck	✓	✓	✓	✓	✓	✓	✓	X
Imandra	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Mythril	✓	X	X	✓	✓	✓	X	X

Fig. Tools / Vulnerabilities Matrix

VII. CONCLUSION

This paper highlights the improvement in the field of blockchain, the havocs with different challenges and openings that are related with these innovations and their straightforward solutions. It is a shared transaction ledger that contains records of exchanges through consensus-based algorithms. Its scope was restricted fairly as there were few transactions but in today's world its scope expanded through utilization of smart contracts in order to computerize different assignments by decreasing their trouble and making them more productive. Blockchain is incorporated with different nodes having a consensus system whereas minings is done for assurity of data consistency and authenticity. There are still some challenges that need to be covered which includes security breach, scalability, increasing market cap size. Some other problems include which programming languages to be used for security maintenance. Making sure of the security of smart contracts is still a top priority of the researchers

because the dealings of transactions made our loss unbearable. Thus, we put about some secure coding methods in this paper to assure the security of these contracts making it safe from hijacking transactions. Still there is a room for vulnerability occurrence as development is still undergoing in this giant technology.

VIII. BIBLIOGRAPHIES

Ahsan Tariq is a student of NED University of Engineering And Technology in the discipline of software Engineering. He completed his Inter from Adamjee Government Science College with A1 grade. He is a MERNStack developer who got 10 months training from Saylani Mass IT Training (SMIT) in the domain of Web and Mobile hybrid App Development. His area of interest in this field includes artificial intelligence, web development and blockchain.

Rehan Mumtaz, a software undergraduate at NED University holds a strong interest in the

chain of development and breaking of the whole loop set of technologies. I am a great fan of cyber-security and love to research about this domain and take this as my passion as my expertise solely lies in Penetration testing(especially web and operating system exploitation), Cryptography & Exploit development. I am looking forward to move ahead with my passion and progress with it in the future InshaAllah

Kabeer Ahmed is an undergraduate student at NED University of Engineering and Technology in Software Engineering. His main areas of interest are Offensive Security like OS Exploitation, Web Exploitation and Defensive Security like Digital Forensics, Cryptography. He has hands-on experience in Data Extraction and web penetration testing. He is keenly interested in making this career in this Cyber Security field.

REFERENCES

- [1] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *Proc. 20th Int. Conf. Financial Cryptography Data Security*, 2016, pp. 79–94.
- [2] M. Wohrer and U. Zdun, "Smart contracts: Security patterns in the Ethereum ecosystem and Solidity," in *Proc. Int. Workshop Blockchain Oriented Softw. Eng.*, 2018, pp. 2–8
- [3] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on Ethereum systems security," *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1_43, Jul. 2020, doi: [10.1145/3391195](https://doi.org/10.1145/3391195).
- [4] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "ReGuard: Finding reentrancy bugs in smart contracts," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. Companion*, May 2018, pp. 65_68.
- [5] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 19.
- [6] Y. Huang, Y. Bian, R. Li, J. L. Zhao and P. Shi, "Smart Contract Security: A Software Lifecycle Perspective," in *IEEE Access*, vol. 7, pp. 150184-150202, 2019, doi: [10.1109/ACCESS.2019.2946988](https://doi.org/10.1109/ACCESS.2019.2946988).
- [7] Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing smart contracts with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 438–453. <https://doi.org/10.1145/3385412.3385982>
- [8] Venkata Siva Vijayendra Bhamidipati, Michael Chan, Derek Chamorro, Arpit Jain, and Ashok Murthy. 2019. Adaptive Security for Smart Contracts using High Granularity Metrics. *Proceedings of the 3rd International Conference on Vision, Image and Signal Processing*. Association for Computing Machinery, New York, NY, USA, Article 83, 1–6. <https://doi.org/10.1145/3387168.3387214>
- [9] Giuseppe Crincoli, Giacomo Iadarola, Piera Elena La Rocca, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. 2022. Vulnerable Smart

Contract Detection by Means of Model Checking. In Proceedings of the Fourth ACM International Symposium on Blockchain and Secure Critical Infrastructure (BSCI '22). Association for Computing Machinery, New York, NY, USA, 3–10. <https://doi.org/10.1145/3494106.3528672>

[10] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 454–469. <https://doi.org/10.1145/3385412.3385990>

[11] Ikram Garfatta, Kaïs Klai, Mohamed Graïet, and Walid Gaaloul. 2022. Model checking of vulnerabilities in smart contracts: a solidity-to-CPN approach. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22). Association for Computing Machinery, New York, NY, USA, 316–325. <https://doi.org/10.1145/3477314.3507309>

[12] Mengjie Ding, Peiru Li, Shanshan Li, and He Zhang. 2021. HFContractFuzzer: Fuzzing Hyperledger Fabric Smart Contracts for Vulnerability Detection. In Evaluation and Assessment in Software Engineering (EASE 2021). Association for Computing Machinery, New York, NY, USA, 321–328. <https://doi.org/10.1145/3463274.3463351>

[13] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for

Computing Machinery, New York, NY, USA, 254–269.

[14] J. Chen, "Finding Ethereum Smart Contracts Security Issues by Comparing History Versions," 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2020, pp. 1382–1384.

[15] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo and T. Chen, "Defining Smart Contract Defects on Ethereum," in IEEE Transactions on Software Engineering, vol. 48, no. 1, pp. 327–345, 1 Jan. 2022, doi: 10.1109/TSE.2020.2989002.

[16] R. Sujeetha and C. A. S. Deiva Preetha, "A Literature Survey on Smart Contract Testing and Analysis for Smart Contract Based Blockchain Application Development," 2021 2nd International Conference on Smart Electronics and Communication (ICOSEC), 2021, pp. 378–385, doi: 10.1109/ICOSEC51865.2021.9591750.

[17] A. Dika and M. Nowostawski, "Security Vulnerabilities in Ethereum Smart Contracts," 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 2018, pp. 955–962, doi: 10.1109/Cybermatics_2018.2018.00182.

[18] E. Zhou et al., "Security Assurance for Smart Contract," 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), 2018, pp. 1–5, doi: 10.1109/NTMS.2018.8328743.

[19] Cryptopedia Staff , "What Was The DAO?" , Cryptopedia, Powered by Gemini, <https://www.gemini.com/cryptopedia/the-dao-hack-makerdao#section-the-dao-hack>

