Blockchain Technology and Vulnerability Exploits on Smart Contracts

Blockchain Technology and Vulnerability Exploits on Smart Contracts

1st Iman Darvishi
School of Computing and Eng
University of West London
London, United Kingdom
iman.darvishi@uwl.ac.uk

2nd Bismark Tei Asare
School of Arts, Hum. and Social Sc
University of Roehampton
London, United Kingdom
bismark.Asare@roehampton.ac.uk

4th Ahmad Musa
School of Eng, Tech. and Design
Canterbury Christ Church Uni
United Kingdom
ahmad.musa@canterbury.ac.uk

1st Abel Yeboah-Ofori
School of Computing and Eng
University of West London
London, United Kingdom
abel.yeboah-ofori@uwl.ac.uk

3rd Waheed Oseni
School of Computing and Eng
University of West London
London, United Kingdom
waheed.oseni@uwl.ac.uk

5th Aishat Ganiyu
School of Eng., Phys.& Math. Sc.
Royal Holloway Uni of London
London, United Kingdom
aishat.ganiyu.2021@live.rhul.ac.uk

***Abstract*— The immutability of smart-contract characteristics is a significant key benefit of blockchain. However, after we deploy a smart contract in a blockchain, we cannot change, modify, or debug it. Further, wrong or vulnerable coding implementation in smart contracts could have error output that may have severe consequences in the future. Thus, the challenge of finding vulnerabilities in the smart contract is vital to stop criminals from performing malicious exploits during Defi transactions. The paper explores Blockchain Technology in Smart Contracts to detect vulnerability exploits focused on general purchase agreements and smart contracts. The novelty contributions of the paper are threefold: First, we explore the existing blockchain vulnerabilities and how attackers exploit decentralized financial transactions (Defi), including re-entrancy attacks, 51% attacks, and double spending issues. Secondly, we set up a Remix virtual platform using the solidity tool to demonstrate a purchase agreement between client and seller that can interact in a smart contract to determine how it can be exploited. The implementations show how the attacker can call the withdraw function recursively before the transaction updates the balance during transaction procedures. Finally, we recommend control mechanisms to improve blockchain security in the purchase agreement and re-entrancy attacks. Our results show that re-entrancy attacks and purchase agreement smart contracts can be secured by developing modifiers to update the bank balance before completing transactions.**

Keywords*— *Blockchain, Smart Contract, Purchase Agreement, Re-entrancy Attacks, Defi Attacks, Solidity

I. INTRODUCTION

Blockchain technology holds vast promise for every business, society, and individual involving the supply chain. The complete form of blockchain that could have a smart contract is Ethereum, a blockchain technology platform. The blockchain operation runs peer-to-peer based on the trust of each component or member of the blockchain. Smart contracts rule the processes and help the transaction to run within a specific and defined frame [1]. Centralising information flow on the Internet is one of the biggest user challenges in terms of interruption; however, communication over the decentralised Internet benefits individuals by having integrity, availability, and confidentiality in place. Each time we need to operate such a transaction, the smart contract can perform a task such as deleting, returning, or modifying the assets [2]. Blockchain can also be used in the business process model, where multiple companies work together to achieve the same objective in the blockchain environment [3]. Although developing such a quality smart contract has attracted many developers in the last few years, the number of attackers stealing people's money by misusing the weakness of smart contracts is increasing, making them so challenging. We explain the re-entrancy attack and suggest a solution to protect smart contract transactions against such attacks. Figure 1 demonstrates the essential steps behind the vulnerable remote purchasing agreement procedures for smart contracts. Blockchain technology is a distributed, immutable ledger that records transactions and monitors assets in a networked corporate environment. Real estate, vehicles, money, and land are all examples of physical assets (intellectual property, patents, copyrights, branding) [2].

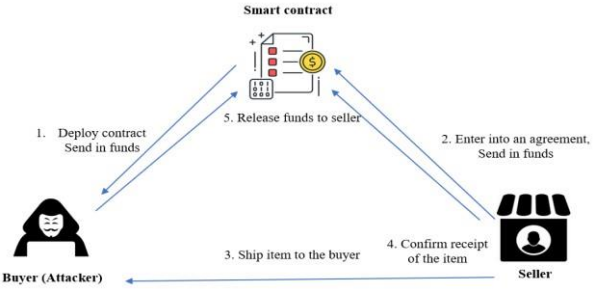

Fig. 1. Vulnerable Remote Purchase Agreement Procedures

There are challenges in the existing purchase agreements leading to various exploits, including data breaches, privacy, and other attacks that have compromised the safety measures employed by these providers [4]. First, the current purchase agreement is vulnerable to the seller since a buyer (attacker) can deny that they received the product or assets they have purchased through the purchase agreement contract; hence, they can refund their money after they have already received their service or product. Secondly, simulate the Ether bank, three clients, and an attacker that attempts to withdraw Ether from the bank recursively by calling the withdraw function, which is a smart bank contract that can let an attacker drain the bank balance. Then, recommend solidity codes to protect our bank from re-entrancy attacks. The blockchain functions use blocks of data created whenever a transaction takes place. These blocks reveal the circulation of an asset, which may be physical (a product) or immaterial (a service) (intellectual). The blocks verify the precise timing and order of transactions, and their encrypted links prohibit tampering with or

interpolating between blocks [5]. A blockchain is an immutable chain in which transactions are stored in stored in a historical manner.

A. *Exchange Transaction in DeFi*

Exchange companies profit from each transaction by facilitating the exchange of assets and performing a transaction. However, in a smart contract, Defi, the seller and buyer exchange cryptocurrency or coin-based assets using only a smart contract in the middle. Here are some benefits: Fewer fractions and less cost.

B. *Recent Blockchain Attacks*

Attackers are always trying to find a way to get into the system and perform their malicious activity. Here are some recent blockchain attacks.

- Defi attack: A 625.5-million-dollar amount that attackers could gain from Axie Infinity's ronin network when they successfully hacked the private key and initiated approval for their fraudulent transaction, FXEmire reported in April 2022 [6].
- DDOS attack on smart contract: Cloudflare reported 15.3 million HTTPS requests for their crypto platform in May 2022; attackers requested 809 million data packets per second and were able to hit the maximum bandwidth of their application. [7].
- Protocol exploits on the qubit finance: Another devastating attack led to 80 million dollars lost for qubit finance when attackers successfully exploited their smart contracts and got away with that [8].
- Blockchain Defi attack on the poly network: The TRM global investigation team reports one of the biggest Defi hacks in history, in which attackers could take 600 million dollars from three different blockchains on a network called poly network [9].

The novelty contributions of the paper are threefold: First, we explore the existing blockchain vulnerabilities and how attackers exploit decentralised financial transactions (Defi,) including re-enteracy attacks, 51% attacks, and double spending issues. Secondly, we set up a Remix virtual platform using the solidity tool to demonstrate a purchase agreement between client and seller that can interact in a smart contract to determine how it can be exploited. Our implementations show that the attacker can call the withdraw function recursively before the bank updates the balance during transaction procedures. Finally, we recommend control mechanisms to improve blockchain security in purchase agreements and re-entrancy attacks. Our results show that re-entrancy attacks and purchase agreements in a smart contract can be secured by developing modifiers that update the bank balance before completing transactions. Further, we have developed modifiers to ensure that the payment transaction is completed for the purchase agreement.

II. Related works

This section discusses the related work of blockchain technology and the state of the art. We consider blockchain technologies and architectures, blockchain transactions, purchase agreements and vulnerabilities. Further, we review some approaches suggested by researchers to cover smart contract issues and perform different tests on smart contract applications[10]. Designed an extensible architecture based on consortium Blockchain by analysing the key technologies and classifications since the scope of application of public and private blockchains is relatively narrow. [10]. [11] proposed a blockchain architecture for industrial applications using frameworks to compare public and permissioned blockchains suited explicitly for industrial applications. [11].[12] proposed architectural design decisions for Blockchain-based applications by systematically exploring architectural design decisions and options in terms of patterns and practices. The paper did not address this [12]. [13] proposed financial data security sharing solutions based on blockchain technology and proxy re-encryption technology by considering solutions that consist of data sharing models using encryption and data sharing protocols using distributed storage, decentralized management and Tamper-proof characteristics of blockchain. However, that could be vulnerable to 51% attack security [13].

Regarding verifications in smart contracts, [14] presented a verification and validation model with a hierarchical process through smart contracts using layers of abstraction, value-added services and authenticity-based AI. The author applied solutions based on distributed ledger technology for the decentralized approach [14]. [15] discussed security challenges and defiance approaches for blockchain-based services, and used ConCERT to conduct a smart contract formal verification experiment by property testing and CVE and CNVD to analyze the vulnerability and enumeration of Alibaba's blockchain services [15]. [16] proposed a tool for mutation testing of Ethereum smart contracts using a blockchain and testing by transforming the smart contract to its source version, which must be in the blockchain's test directory. Finally, this test tool has a friendly user interface that is graphical and easy to use and allows users to test the smart contract in solidity [16].

Regarding vulnerabilities in smart contract codes [17], a test on smart contracts was performed to detect the vulnerabilities that have led to many financial losses for business application users. The authors focused on detecting smart contract vulnerabilities to secure the code or functional flaws since existing tools that are in use are decreasing the performance of codes to be analysed and are constantly being rewritten and proposed by Eth2Vec. This machine-learning-based static analysis tool detects smart contract vulnerabilities [17]. Furthermore, [18] compared two different blockchain technologies, SBlockchain and TBlockchain, in smart contract management systems by developing a framework for a blockchain-based smart contract and transaction management system on a Decentralized Autonomous Organization and enterprise levels. The smart contracts are maintained in the SBlockchain, while the data produced by the smart contracts are kept in the TBlockchain. [18]. Regarding blockchain application development, [19] examined the methods and approaches covered in related papers in levels of testing and analysis for smart contract-based blockchain application development with the purpose of contracting for electronic agreements to support the functions [19].

All the existing literature is relevant and contributes to knowledge and research in blockchain technology and vulnerability exploits. For instance, [10] implemented a testbed for functional, security, and performance testing to analyze the extensible consortium blockchain framework in changing scenarios and needs. [15] , used the ConCERT approach to test a smart contract verification experiment on Alibaba's blockchain services and CVE and CNVD for vulnerability analysis and enumeration. Further, [17] proposed Eth2Vec, a machine-learning-based static analysis tool that detects smart contract vulnerabilities. The Eth2Vec maintains its robustness against code rewrites. However, it is susceptible to attacks. [14] applied verification and validation using layers of abstraction, value-added services, and authenticity-based AI solutions models based on distributed ledger technology for real-data marketplace applications and transactions using the decentralized model. [11] Implemented a solution based on Ethereum for proof-of-authority by using a consensus algorithm to instruct the running procedures in the source code

of the smart contract, which is characterized by a set of validator nodes running the blockchain to ensure transparency and immutability. However, the authors did not focus on cryptocurrency and tokens. Furthermore, [18] developed a framework that compared SBlockchain and TBlockchain technologies in smart contract management systems. [13] Proposed encryption methods and data-sharing protocols using distributed storage, decentralized management, non-tampering characteristics, and proxy re-encryption for the proof-of-stake algorithm that realizes data sharing among users for blockchain security.

However, no one has set up a Remix virtual platform using the solidity tool to demonstrate a purchase agreement between client and seller that can interact in a smart contract to determine how it can be exploited. The paper explores Blockchain Technology in Smart Contracts to detect vulnerability exploits, focusing on general purchase agreements and smart contract transactions.

III. Approach

This section discusses the approach used for the implementation process for blockchain technology and vulnerability assessment in smart contracts. We applied a qualitative [19] approach and secondary data to understand the functionality of the smart contract and consider its vulnerability. Further, we deployed an attack on a single solidity smart contract to detect any weakness during transactions. The purpose is threefold: first, to describe the cascading cyberattack deployed on the blockchain; second, to highlight the vulnerabilities of both purchase agreements and the bank's smart contracts; and third, to offer a solution to cover the current loopholes in the coding of both purchase agreements and the smart contracts used by Ethereum banks. The implementation attempts to address the following challenges. First, the seller is at risk under the terms of the existing purchase agreement since the customer (the attacker) might claim they never got the service or goods for which they paid under the terms of the purchase agreement contract and demand a refund. Second, we will simulate an Ether bank with users and an attacker who will try to take Ether from the bank in a recursive manner by repeatedly using the withdraw function to see if we can deplete the bank's funds. Then, solidity codes should be suggested to prevent re-entrancy attacks.

IV. IMPLEMENTATION

This section considers implementing a smart contract to facilitate safe remote purchases from re-entrancy attacks where the buyer and seller are protected without needing a centralised trusted authority. Such a contract might assist clients in purchasing products from the EOTORO platform.

- Step 1: Safe purchase agreement. The seller publishes the contract and sets the value for the item for sale
- Step 2: safe purchase agreement. The buyer sends funds to the contract, which puts the transaction into a secure lock state, freezing the funds for the time being.
- Step 3: Safe purchase agreement. The seller ships the item for sale to the buyer.
- Step 4: Safe purchase agreement. When buyers receive the item, they invoke a confirmation of the smart contract to acknowledge the shipment receipt.
- Step 5: Safe purchase agreement. Finally, smart contracts release the locked funds back to the seller after receipt of shipment confirmation.

The security problem of safe remote purchase is that the current simple design of the smart contract has a Security flaw. That security vulnerability is exploited when we rely on the buyer's honesty to say whether they have received the item. What if the buyer never reports that the item has been received, and the seller never receives their money?

A. Implementations Tools Used

We used a Solidity tool for put implementations. The Solidity tool is a high-level object-oriented language that implements the smart contract. The syntax is quite similar to Java scripts. It runs on an Ethereum virtual machine (EVM) with other development tools, including Truffle suit, hardhat, remix, Ganache, and many others that assist in building smart contracts. We use a high-level program for the Smart contract and Defi attacks that compiles EVM bytecode and deploys it to the blockchain for execution. Further, it is immutable, indicating that once the transaction has been successfully mined and sent out, there is no way to revert it [20]. It is difficult to trace once the attacker gets into the mixer during the torrential cache, leading to Admin key compromise, Private key compromise, computer trojan, Phishing attack and Malicious insider. Malicious insider: in Defi projects, people share admin keys, and a malicious insider can use that to call admin functions and transfer all the tokens out [21].

B. Variables to Hold the Item's Value

We need variables to hold the item's value for sale or the sale price. We will need address types for the seller and the buyer, and we also need variables to hold the state of the contract or the state of the purchase. In other words, to differentiate the different states, we consider the following: Has the buyer paid yet, received funds, or released funds?

Functions: First, we will need some way for the seller to initially send money into the contract, which the constructor or a dedicated function could handle. We will need a function to confirm the purchase. That will indicate when the buyer has sent money into the contract; we will also need another operation to confirm receipt of the item for sale, which the buyer will call. Eventually, we will need functions to pay the seller once the deal is done. Additionally, we might need to add a function to abort the mission, allowing the cancellation of the entire agreement. This function should only be callable before the buyer has sent money into the contract.

C. Implementation Tools Remix Editor

This section discusses the implementation tool for our work. We consider the Remix editor a tool because it is an in-browser IDE. It supports trial compilation and deployment during implementation in an isolated environment before the smart contract is pushed into the cloud. We use Remix to compile and deploy our Solidity contracts and test them quickly without installing infrastructures such as truffle or hardhat.

D. Implementing Purchase Agreement in Smart Contract

The default workspace folders in the file explorer section contain some premade contracts by remix in Solidity. By selecting any contract, we can see its content on the right-hand side. We expanded the sample size of the Solidity folder.

E. Solidity Coding

To begin the solidity coding, we start by selecting a contract file from the file explorer, removing all the codes from the file, and then renaming the file to "PurchaseAgreement.sol". *Sol tells us the file contains solidity codes at the end of every solidity file*. Now, we are ready to start our smart contract transactions.

Step 1: We set the SPDX identifier; we will first add the license identifier. (//SPDX-License-Identifier: MIT)

Step 2: We indicate the version of our smart contract by choosing the related compiler version using the *pragma solidity 0.8.11*) code.

Step 3: we have defined our contract class and named it *"Purchase Agreement"*, which describes the attributes such as sellers, buyers, amount, shipment confirmation and wallet address by using the code *(contract PurchaseAgreement {}).*
Step 4: we define a variable for an item by using the code *(unit public value;)* to store a value that will hold the item for sale. The "uint" in the code shortens the form to "un256".
Step 5: illustrates how we define the address of the wallets by typing the commands *(address payable public seller;)* and *(address payable public buyer;)*
Step 6: we define some variable to hold the state of the contract at any given time by using the code *(enum State { Created, Locked, Released, Inactive })* to indicate the state, and for that, we use "ENUM",
Step 7: We create a state variable using the specified ENUM state type. The state (public state) will be a public variable used to interact with other transaction components.
Step 8: we have not assigned any value to the state variable, at least upfront; in this case, by default, this variable will be initialised with the initial value of the values we have specified in the ENUM so that it will be created by default. Figure 4 illustrates the constructor function and its variables:
Step 9: The "confirm purchase" function enables the buyer to send in money and be designated as the buyer for the contract's life. We use the function keyword and then confirm the purchase, which is an external function since the buyer must be able to invoke this function outside the smart contract code. Further, we updated the contract state since, by default, we assigned the state to create the mode. Hence, we need to lock the contract once the buyer's payment to the seller has been confirmed.
Step 10: Create a couple of modifiers to facilitate those checks. We used the modifier as a separate function or entity to add here to qualify this function and restrict this function to satisfy the terms of the modifier. To do that, first, we must tackle the requirement that the buyer send twice the amount of the purchased item to proceed with the transaction. Here, the value is the item price that the seller has set, so the transaction should be equal to two times greater; if that evaluates to false, we will send back an error message and say: "Please send in 2X the purchase amount". To output an error message if the deployer calls this function while the state is in a different mode, we use this code: (*error InvalidState()*. Further, we check the state of the contract, so for that, we use a modifier, and the modifier can call the revert function to revert the transaction if the condition is not met. Then, the revert function will return a call on a custom error.

F. Create the Modifier

We create our modifier and adjust its conditional statement to function if we want to revert the transaction. This method inputs the state type as an argument and state with the underscore to differentiate that this is an argument from our modifier. To ensure that the state is not equal to the state argument we are questioning, we check the modifier type; then, if the state is invalid, we revert the function and call the custom error we previously defined. In the end, there is an underscore and semicolon, a placeholder for executing the rest of the function that the modifier applied. So, to use this to confirm the purchase function by placing it after the external modifier and before the payable function, finally, we have to input the state argument by using this code: (*function confirmPurchase() external instate payable {*). Re-entrancy attacks, or Ethereum heists, are among the most destructive attack vectors where malicious smart contracts can drain all the funds from the victim contract. This attack can function with a recursive call from an external function to the victim's contract withdrawal part. We run a sample of a re-entrancy attack to explore its procedure and go through a few different ways that we can protect our code attack.

G. How a Re-entrancy Attack Works

A re-entrancy attack happens when malicious contracts that we call attackers contracts call victim contracts in case they gain more control over code execution than was ever intended, disrupting the intended state of the victim contract and manipulating it in unexpected ways. For instance, the attacker can call a withdrawal function on a victim contract, which then sends funds to the attacker. Still, the attacker then gains control of code execution via its fallback or receives the part and can recursively call the victim's withdrawal function repeatedly before the victim can update its account balances to reflect the withdrawal. That continues until the attacker has effectively drained the victim of all its funds. Figure 2 demonstrates the attacker and victim re-entrancy attack:

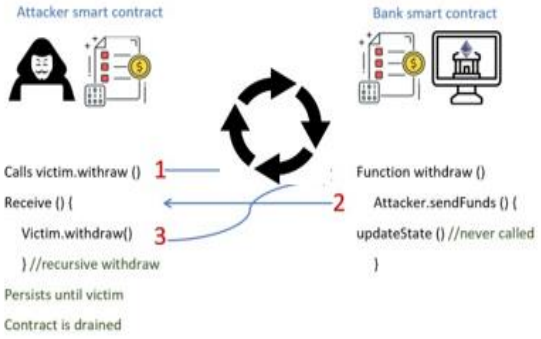

Fig. 2. Attacker and Victim Re-entrancy Attack

We used the same attack in the DAO attack 2016, in which 60 million dollars in Ether was stolen, resulting in the controversial forking of the Ethereum blockchain into Ethereum and Ethereum Classic to return the stolen funds. We demonstrate the attack using the remix Ethereum editor online platform. In our remix editor, we have two smart contracts:

- Victim: Ether bank's smart contract (EtherBank.sol)
- Attacker code smart contract (Attacker. sol)

H. Ether Bank Code Smart Contract

Step 1: In the ether bank savings account, there is a mapping of addresses to balances that keep track of all the funds in the bank. Figure 3 demonstrates the smart contract:

```
contract EtherBank is ReentrancyGuard {
    using Address for address payable;

    // keeps track of all savings account balances
    mapping(address => uint) public balances;
```

Fig. 3. Bank Smart Contract

Step 2: The deposit function, an external payable type, includes the "*msg. sender*" related to the valet address, and the value is the amount the client will deposit. The deposit function allows users to send in some amount of Ether to update the balance. The withdraw function indicates the withdrawal amount and could be the entire account balance. Figure 4 shows the withdraw function from the external type that functions until there is no available balance.

```
// withdraw all funds from the user's account
function withdraw() external nonReentrant {
    require(balances[msg.sender] > 0, "Withdrawl amount exceeds available balance.");

    console.log("");
    console.log("EtherBank balance: ", address(this).balance);
    console.log("Attacker balance: ", balances[msg.sender]);
    console.log("");

    payable(msg.sender).sendValue(balances[msg.sender]);
    balances[msg.sender] = 0;
}
```

Fig. 4. Withdraw Function

We discuss the logging, attacker, and constructor functions of the Logging Function: We need logging as it is the function that the attacker will call recursively until the entire smart contract balance is drained. With this function, we can check the balance at any time during our testing to track the process. We use the"*getBalance*" function to input the item name and output the amount defined for the item.

Attacker Contract: The attacker contract will be called the Ether bank contract, so to do that, we define an interface with the functions that will be reached from the Ether bank contract. Constructor Function: The constructor function assigns the payment values to the seller, and the owner variable is associated with the seller's wallet address, designating them as the owner of the resources. We pass the contract address that interacts with our contract and call the functions to proceed with the transaction. We also set the owner variable since we have some of these functions restricted to the owner only, as discussed in the constructor function in Figure 5:

```
    // check the total balance of the Attacker contract
    function getBalance() external view returns (uint) {
        return address(this).balance;

contract Attacker {
    IEtherBank public immutable etherBank;
    address private owner;

    constructor() payable {
        seller = payable
        owner = msg.sender;
    }


    function attack() external payable onlyOwner {
        etherBank.deposit{value: msg.value}();
        etherBank.withdraw();
    }

    receive() external payable {
        if (address(etherBank).balance > 0) {
            console.log("reentering...");
            etherBank.withdraw();
        } else {
            console.log('victim account drained');
            payable(owner).transfer(address(this).balance);
        }
    }
```

Figure. 5. Attacker Contract

J. Attack Function

First step: the attacker depositing into the bank, and the reason for that is because the withdraw function checks to make sure that there are some balances associated with the address invoking withdraws, so we have to be a member of the bank and have deposited some of the funds to withdraw. The code in Figure 6 demonstrates the attack function:

```
function attack() external payable onlyOwner {
    etherBank.deposit{value: msg.value}();
    etherBank.withdraw();
}
```

Fig 6: Attack Function

Further, the attacker function immediately invokes the withdraw function; the withdraw function sends the Ether to the message sender, which is the account gathering the withdrawal in the Ether bank's smart contract. Using code: "*payable(msg.sender).sendValue(balances[msg.sender];*" Furthermore, it updates the balances mapping to reflect that withdrawal; however, what will happen is that in the receive function of the smart contract, there is a smart contract call back that executes whenever they receive any funds, and here is where the attacker gains control over execution and can call withdraw again and again. They can do this because the victim has not yet updated the balances to reflect the withdrawal, so she can pass the requirements check and recursively call withdrawal. In the receive function, the code: (if (address(etherBank).balance > 0) {) is used to perform a check to see if the bank has any money at all. Moreover, as long as it does, it will keep on recursively calling withdrawal, and once it is completely drained, it transfers all the funds from the attacker's smart contract to the attacker's wallet, which is owned. From that point, we log in and see the account as the *victim's account drained*. Figure 7 indicates there is no money in the victim's account.

```
    } else {
        console.log('victim account drained');
        payable(owner).transfer(address(this).balance);
    }
```

Fig. 7. Victim account drained.

V. Results and Discussion

This section demonstrates how we implement the Solidity tool in a virtual environment to detect vulnerabilities and deploy an attacker during smart contract transactions. This procedure illustrates how an attacker can exploit the vulnerable current of most of the bank's smart contracts by depositing some Ether, draining all the bank balance and transferring it into their account. To simulate the bank, we have a bank with three clients. Each client deposits 10 Ethers to the bank from a different account number. The attacker will pretend he is the fourth customer by depositing 2 Ether in the bank and attempting to withdraw his money.

Right before the bank updates its balance, the attacker will withdraw 2 Ether from the bank every time until the bank balance is 0. To achieve that, we follow the procedure below:
Simulate the bank interaction by having three clients

- Three times deposits of 10 Ether from 3 different accounts for each client to the bank.
- Deploy attacker smart contract to drain the bank balance

A. Phase 1:

Compile the Bank's Smart Contract: To compile the bank's smart contract, first, we select both "Etherbank. sol" and the attacker.sol contracts. Then, we select the same version for the compiler we implemented in our ether bank's smart contract. Further, we must ensure they are the same version; here, we use the 0.8.11 version.

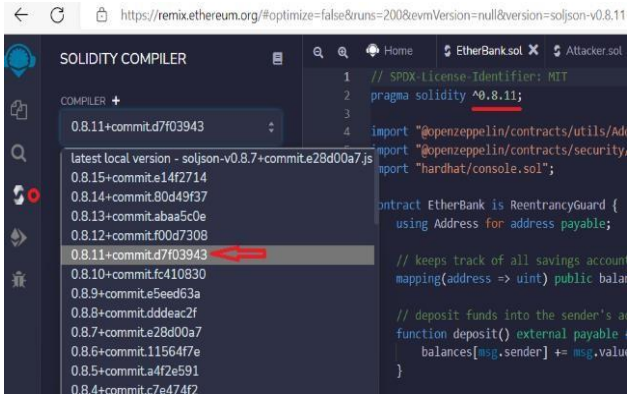

Fig. 8. Compile the Attacker and the Bank Smart Contract

Further, we allocate an account number to our bank, as highlighted in Figure 8.

B. Phase 2: Deploy our Ether Bank's Smart Contract

First, the transferred amount will be reverted to the sender's account after we deploy the smart contract illustrated in Figure 9; we have access to the function that we defined

earlier with solidity; we will have three different clients' next steps. Each client will deposit 10 Ether to the bank, and we can check the balance to monitor the customer's balance every time.

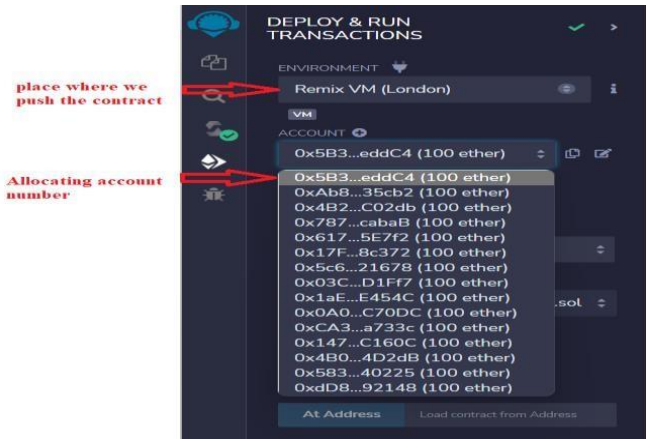

Fig. 9. Deploy Phase for Smart Contract

First, we check the bank account balance simply by using the balance button that calls the balance function from our bank's smart contract. The balance is '0' as no client has yet deposited. To deposit as a client, we select the currency type to Ether, choose 10 Ether to deposit and finally press the deposit button that calls the deposit function from our solidity code. Now, we assign the account number to a customer. We transferred the deposit value to 10 and selected Ether as a currency. Finally, solidity calls the deposit function that increases the account balance by adding the current ratio, which is zero, with the amount the client sends, which is 10 Ether.

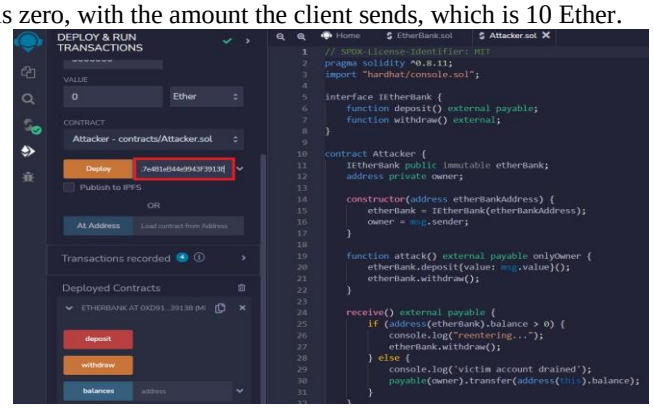

Fig. 10. Attacker Smart Contract Deposit

Further, the deposit function transfers the amount to the client account number, as shown in a red rectangle at the top left of Figure 10. The bank has one customer who has deposited 10 Ethers from their valet address into the bank. We have also assigned our customer an account number. We added two more customers and deposited 10 Ether for each to make it more accurate. We deposited Ether from the second customer account. Furthermore, we changed the account number and assigned a new account number for our second client. Then, we had 10 Ether to deposit in our bank. We will call the deposit function in the bank's smart contract to do this; we repeated the deposit operation for the last customer again and assigned a new account number to our third customer. Again, we will specify 10 Ethers for them to deposit into their account in our bank by calling the deposit function. We have deposited 30 Ether from 3 clients with three other bank account numbers to our bank. We can call the balance function to check how much Ether we have in our bank. Phase 3: compile and deploy the attacker's smart contract. We selected our attacker's smart contract to compile our attacker's contract.

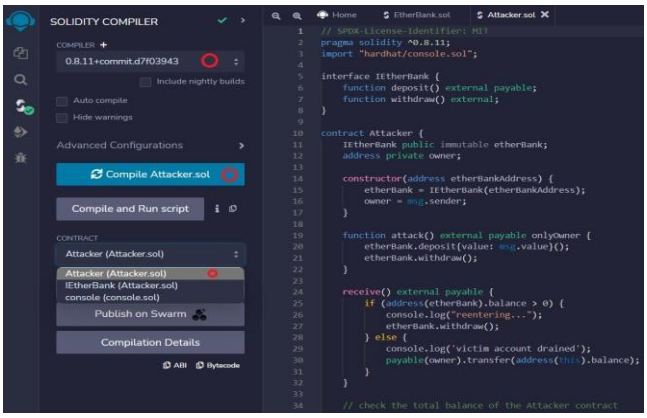

Fig. 11. Attacker Smart Contract is Written in Solidity Language

Figure 11 depicts how we set the compiler version to 0.8.11 to compile it successfully and select the attacker contract from the contract menu. The green check mark beside the compile menu illustrates that we have compiled our smart contract with no issues or errors.

C. Deploy the Attacker on the Smart Contract

Deploying the smart contract attacker is different from a bank's smart contract because, for the attacker, we have to target our bank valet address, and all the attacker's smart contract functions have to interact with the bank account directly. The reason is that if there is no account number assigned to the bank or no cryptocurrency in the bank, then there is no point in initiating an attack. To use the bank valet address, we copied the bank valet address and pasted it into the field box for the attacker's deploy section, then pressed the deploy button as in Figure 12. We show how we first choose the attacker's address. The smart contract selects the bank's smart contract. We paste the address into the deployed test and use the copier feature to copy the bank valet field.

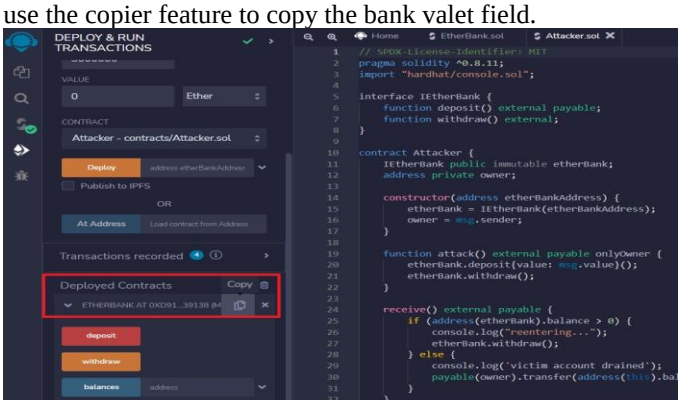

Fig. 12. Deploy the Attacker Smart Contract

In Figure 13, we have copied the bank valet address to the contract that will appear under the bank's smart contract. Deploy our smart contract on it, then press deploy.

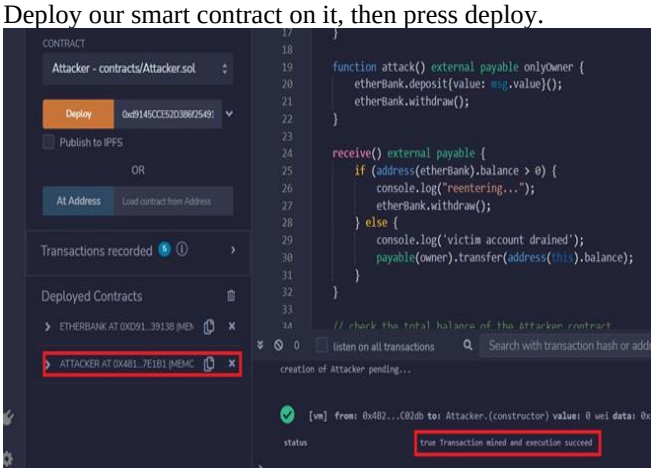

Fig 13. Assign the Bank Valet Address to Deploy our Smart Contract

Once it successfully deploys the attacker contract, the rectangular red box on the right side, which is the compiler output, indicates that the operation has been successful. As we mentioned at the beginning of chapter four, to attack the bank, the attacker first deposits some Ether to establish trust. Then, with its withdraw function in the loop, the attacker takes money from the bank until the account balance is zero.

D. Phase 4: Attack and Result

We deposited two Ethers to the bank; select 2 Ethers, then press deposit. Figure 14:

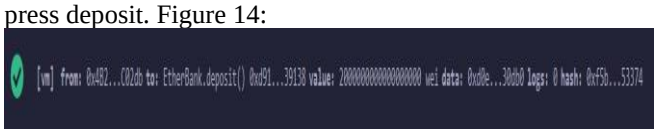

Fig. 14. Deposit Two Ether to the Bank

Figure 14 illustrates the successful compiler output. Now, we will clear the output and select 2 Ether to start; we have deposited 2 Ether.

Fig. 15. The Results Show that the "Victim Account is Drained"

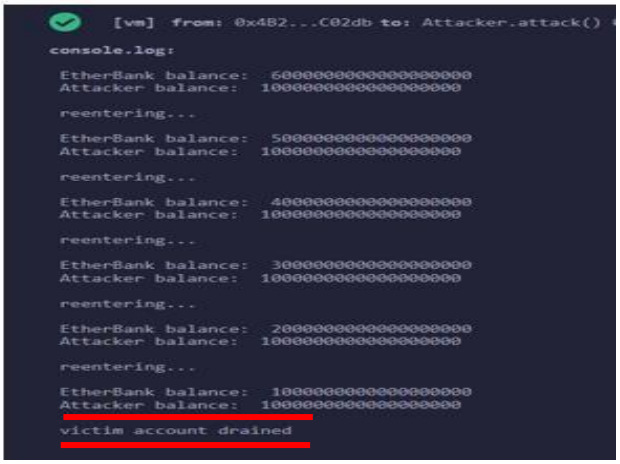

Figure 15 shows the total balance is 6 Ether on the red mark number 1, and the attacker took 1 Ether from the bank. Before the bank updated its balance again, the attacker took another 1 Ether from their bank balance every time. Notice that this back balance, which we can see in this figure, has not been obtained from the bank. This is the re-entrancy function in the attacker contract calculating it. In our case, this re-entrancy functions 6 times until the bank balance is zero, which shows the victim's account drained in Figure 15. Further, the last six re-entrancy attacks until the bank balance has entirely drained. If the amount of the Ether that the attacker will drain in every call of the withdraw function from the attacker contract is greater than the bank balance, we have still drained the bank balance. We have redone all the operations from Phase 1 to Phase 4 to get this message.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
abstract contract ReentrancyGuard {
    uint256 private constant NOT_ENTERED = 1;
    uint256 private constant ENTERED = 2;
    uint256 private _status;
    error ReentrancyGuardReentrantCall();
    constructor() {
        _status = NOT_ENTERED;
    }
    modifier nonReentrant() {
        _nonReentrantBefore();
        _;
        _nonReentrantAfter();
    }
    function _nonReentrantBefore() private {
        // On the first call to nonReentrant, _status will be NOT_ENTERED
        if (_status == ENTERED) {
            revert ReentrancyGuardReentrantCall();
        }
        _status = ENTERED;
    }
    function _nonReentrantAfter() private {
        _status = NOT_ENTERED;
    }
    function _reentrancyGuardEntered() internal view returns (bool) {
        return _status == ENTERED;
    }
}
```

Fig. 16. Control Measures for re-entrancy attack

E. Control Measures of Re-entrancy Attack

We discuss how to protect mechanisms against reentrancy attacks. This part of the contract is the best example of implementing a defence against a re-entrancy attack where we have to update the state of the contract and also send funds; the first step here would be updating the contract state to avoid someone invoking this contract again and again and sending funds before the states updated shown in Figure 16.

F. Control Measures of Safe Purchase Agreement

```
1  Modifier OnlyBuyer() {
2      If(msg.sender != buyer) {
3          Revert OnlyBuyer();
4      }
5      _;
6  }
7  constructor() payable {
8      seller = payable(msg.sender)
9      value = msg.value / 2
10 }
11 function paySeller() external onlySeller inState(state.Released) {
12     state = state.Inactive;
13     seller.tranfer(3 * value);
14
15 }
16 Modifier OnlySeller() {
17     If(msg.sender != Seller) {
18         Revert OnlySeller();
19     }
20     _;
21 }
22 function abort() external onlySeller inState(state.Created) {
23     state = state.Inactive
24     seller.tranfer(address(this).balance)
25 }
```

Fig. 17. Control Measures for Safe Purchase Agreement Re-entrancy

In the Safe Purchase Agreement smart contract, we will make a function and call it to confirm receipt; it should be an external function and does not need to be payable as we are not receiving the money in this function. Here, the first step we will take is to update the state. Here is the state of the contract: we release the funds and transfer them to the buyer, but this is not paying the seller; indeed, it is just only returning the deposit to the buyer, and we will need a separate function for the seller, here we need to check two factors: First, we must ensure that only the buyer can invoke the function. Second, we need to ensure that the state is in the locked position or locked status; although we have left it locked after confirming the purchase, we must check to ensure the contract form is correct to invoke this. We can use the state modifier we have already set up to handle that. Next, we need to set up a new modifier and a new custom error to ensure the buyer is the only one who can call this function; we will implement it right below the first custom in the smart contract. First, we will do the error message "Only the buyer can call this function" The error code for that would be: "error OnlyBuyer;" The next step is to create a new modifier state that if the sender of this transaction is not the buyer, then we revert the function and call our only buyer custom error message; lines 1 to 6 in the Figure 15 illustrates the code for revert modifier. Before we deploy and test it, we need to set the value in the constructor, so the value should be divided by two since we are sending it twice; the new constructor function is implemented in lines 7 to 10. Now, we need to pay our seller once the condition of the contract or the sale has been satisfied; we will implement a function to pay the seller. This function needs to be external, and only the seller can invoke this function, so we need a modifier similar to the one we just set up for the seller. We will also need to recheck the state, and this time, we need to set the state in the release mode; we also need to set up a modifier containing a custom error; the error code is "error OnlySeller()". Further, updating and setting the state to inactive is essential to prevent a reentrancy attack. Then we have to send funds back to the seller; hence, the seller will first receive the price of the item sale, and then he will also accept that same amount twice to

represent his initial deposit. Hence, we are going to sell three times his value. Lines 16 to 21 contain the code for the pay-to-seller function. The modifier implemented determines that the seller is the only participant who can invoke the pay function; otherwise, the transaction will be reverted.

The final function will safeguard the seller if they must abort the mission and back out of the transaction. We only want to allow that to happen if we are in the beginning stage, so before the buyer has sent any money in, otherwise that would not be entirely fair for the buyer to lock up his funds and cancel everything. Hence, they can only call this function if we are in the state that created status. So, after completing the abort function, we will update the form as we always do; we inactive the state and ask for a money refund.

The aborted function operates only if the buyer has not paid yet. If the seller decides to cancel and revert the transaction, this function will cancel the whole operation.

VI. Conclusion

The paper conducts a vulnerability assessment of smart contracts to find their weaknesses. Since attackers are always trying to find a new way to get into the system and act maliciously to achieve their purposes. We focused on exploiting vulnerabilities in smart contracts using the solidity code to prevent persistent re-entrancy attacks by implementing modifiers that allow us to update the bank balance right after each transaction. We have simulated the smart contract transaction to demonstrate the bank, client and the attacker's smart contract. First, we deposited some Ether to the bank from the client's account and updated the bank balance. Second, we deposited some ether from the attacker's account into the bank and updated the balance. Third, we have withdrawn the same amount from the bank to the attacker's account. Fourth, we have withdrawn money from the bank before updating its balance, so we could call the withdrawn function in the attacker's smart contract recursively until the bank balance is fully drained. Figures 24,25 and 26 illustrate the re-entrancy attack. We have implemented and deployed a modifier to confirm the shipment status and send it to the smart contract. In this case, we have ensured the seller gets paid fully and safely, and the buyer cannot revert the transaction.

Future works will consider smart contract vulnerability detection using attack datasets and AI for threat predictions.

References

[1] Daria A. Snegireva (2021) 'Review of Modern Vulnerabilities in Blockchain Systems', in 2021 International Conference on Quality Management, Transport and Information Security, (IT QM IS). doi:10.1109/ITQMIS53292.2021.9642862.

[2] Yang, X., Chen, Y. and Chen, X. (2019) 'Effective Scheme against 51% Attack on Proof-of-work Blockchain with History Weighted Information', in 2019 IEEE International Conference on Blockchain (Blockchain). doi:10.1109/Blockchain.2019.00041.

[3] Wang, S.-H., Wu, C.-C., Liang, Y.-C., Hsieh, L.-H., and Hsiao, H.-C. (2021) 'ProMutator: Detecting Vulnerable Price Oracles in Defi by Mutated Transactions', in 2021 IEEE European Symposium on Security and Privacy Workshops (EuroS PW). doi:10.1109/EuroSPW54576.2021.00047.

[4] A. Yeboah-Ofori, S. K. Sadat and I. Darvishi, "Blockchain Security Encryption to Preserve Data Privacy and Integrity in Cloud Environment," 2023 (FiCloud), pp. 344-351, doi: 10.1109/FiCloud58648.2023.00057.

[5] Leible, S., Schlager, S., Schubotz, M. and Gipp, B. (2019). A Review on Blockchain Technology and Blockchain Projects Fostering Open Science. Frontiers in Blockchain, 2. doi:10.3389/fbloc.2019.00016.

[6] Sinclair, S. (2022). DeFi Exploits Top $1.8B YTD, Though Security 'Getting Better' Immunefi Says. [online] Blockworks. Available at: https://blockworks.co/news/defi-exploits-top-1-8b-ytd-though-security-getting-better-immunefi-says.

[7] Essaid, M., Kim, D., Maeng, S.H., Park, S. and Ju, H.T. (2019) 'A Collaborative DDoS Mitigation Solution Based on Ethereum Smart Contract and RNN-LSTM,' in 2019 (APNOMS), pp. 1–6. doi:10.23919/APNOMS.2019.8892947.

[8] Melinek, Sánchez-Gómez, N., Morales-Trujillo, L. and Torres-Valderrama, J. (2019) 'Towards an Approach for Applying Early Testing to Smart Contracts':, in Proceedings of the 15th International Conference. doi:10.5220/0008386004450453.

[9] Hirtenstein, A. (2021). Crypto Hackers Stole More Than $600 Million From DeFi Network, Then Gave Some of It Back. Wall Street Journal. [online] 11 Aug. https://www.wsj.com/articles/poly-network-hackers-steal-more-than-600-million-in-cryptocurrency-11628691400.

[10] L. Ni, S. Zhang, G. Li, K. Han and H. Sun, "A Design of Extensible Architecture Based on Consortium Blockchain," 2022 IEEE 14th International Conference on Advanced Infocomm Technology (ICAIT), Chongqing, China, doi: 10.1109/ICAIT56197.2022.9862749.

[11] L. Marchesi, M. Marchesi, R. Tonelli, M. I. Lunesu, A blockchain architecture for industrial applications, Blockchain: Research and Applications, Volume 3, Issue 4, 2022, https://doi.org/10.1016/j.bcra.2022.100088.

[12] M. Wöhrer, and U. Zdun. Architectural design decisions for blockchain-based applications. 2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), pp. 1-5.

[13] Z. Su, H. Wang, H. Wang and X. Shi, "A Financial data security sharing solution based on blockchain technology and proxy re-encryption technology," 2020 IEEE doi: 10.1109/IICSPI51290.2020.9332363.

[14] W. Serrano, Verification and Validation for data marketplaces via a blockchain and smart contracts. Blockchain: Res. Appl. 2022, 3, 100100. https://doi.org/10.1016/j.bcra.2022.100100.

[15] H. Chen, X. Luo, L. Shi, Y. Cao, Y. Zhang, Security challenges and defence approaches for blockchain-based services from a full-stack architecture perspective, Blockchain: Research and Applications (2023), doi: https://doi.org/10.1016/j.bcra.2023.100135.

[16] Li, Z., Wu, H., Xu, J., Wang, X., Zhang, L. and Chen, Z. (2019) 'MuSC: A Tool for Mutation Testing of Ethereum Smart Contract', doi:10.1109/ASE.2019.00136.

[17] Ashizawa, N., Yanai, N., Cruz, J.P. and Okamura, S. (2022). Eth2Vec: Learning contract-wide code representations for vulnerability detection on Ethereum smart contracts. Blockchain: Research and Applications, p.100101. doi:https://doi.org/10.1016/j.bcra.2022.100101.

[18] Muneeb, M., Raza, Z., Haq, I.U. and Shafiq, O. (2021). A Blockchain-based Framework for Smart Contracts and Transaction Management. IEEE Access, doi:10.1109/access.2021.3135562.

[19] Sujeetha, R. and Deiva Preetha, C.A.S. (2021). 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). doi:10.1109/icosec51865.2021.9591750.

[20] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo and T. Chen, "DefectChecker: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode," in IEEE Transactions on Software Engineering, 2022, doi: 10.1109/TSE.2021.3054928.

[21] Ajayi, Oluwaseyi & Saadawi, Tarek. (2021). Detecting Insider Attacks in Blockchain Networks. 10.1109/ISNCC52172.2021.9615799.