Smart Contract Security: Auditing a Beginner Smart Contract

Tadija Ciric

10 December 2022

Abstract

Abstract Smart contract technology is an evolving field reshaping modern-day business. The automatic enforcement of contractual terms removes the need for a third party. It enables decentralized trade, which cuts down on administration and service costs, further improving efficiency and reducing risk. Although smart contracts have the potential to drive a new wave of innovation in business processes, they face several challenges. To be reliable smart contracts need to be secure. This paper reviews the security principles, possible vulnerabilities, and solutions of smart contracts. Being the key component of blockchain, the security of smart contracts is critical to furthering the field of decentralized finance and distributed ledgers.

Keywords: *smart contract; security; application; blockchain; Ethereum*

1. Introduction

This paper aims to provide a systematic overview of the security of smart contracts. I will briefly introduce the history of smart contracts and their benefits over standard contracts. I will systematically present relevant information regarding smart contracts and how they relate to blockchain technology. Further, I will describe why smart contract security is important and what it consists of. I will provide an analysis of several smart contract vulnerabilities that are common in the industry and provide up-to-date solutions. Additionally, I will implement the mentioned solutions in my contract. The following are some of this paper's contributions:

- An overview of smart contracts and blockchain properties
- Analysis of crucial smart contract vulnerabilities
- Analysis of solutions to mentioned vulnerabilities
- A sample smart contract with little security
- An audit of the sample smart contract

1.1. *Smart Contract Overview*

Smart contracts are contracts deployed on the blockchain enforced by code that execute automatically once the pre-written conditions are met and verified. The idea of smart contracts was introduced in the 1990s by Nick Szabo, with the idea of having a contract in which the parties can observe the performance and verify whether the contract has been performed or breached [1] [2]. The development of smart contracts led to three improvements over standard centralized contracts:

- Removing the third party from transactions

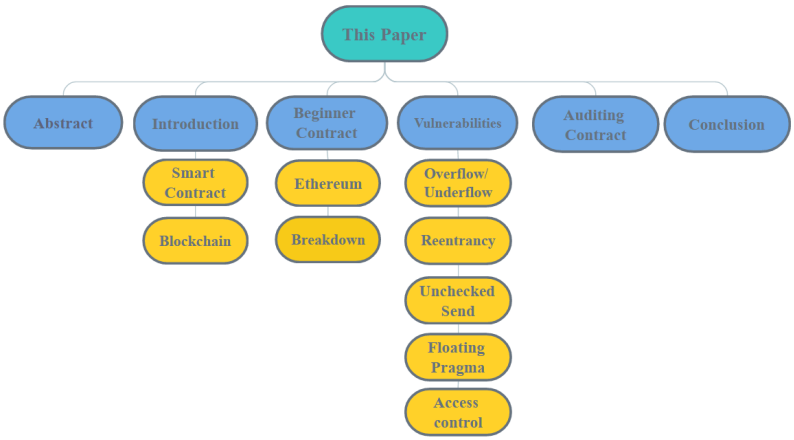

Figure 1. A visual representation of this paper's organization

- Being transparent and secure

- Self-enforcing

Conventional contracts must be completed by a trusted third party in a centralized manner, resulting in lengthy execution and additional costs. As soon as a condition in the contract is fulfilled, the contract is executed, achieving peer-to-peer relations. A feature of the blockchain is that whenever a smart contract is deployed or completed, a new block is added to the blockchain [3]. Because hashes link all blocks, they are immutable and irreversible. All transactions on the blockchain are publicly available, making smart contract usage very secure and transparent [4]. Smart contracts reflect a lack of trust in human behavior, as humans are only sometimes trustworthy. Code has no bias, unlike human factors like governments, bankers, or loaners. Human error would be removed, erasing any discrepancies.

Benefits of smart contracts are [5]:

1) Efficiency and accuracy

When a condition is met, the contract is immediately executed. Because smart contracts are digital and automated, there is no paperwork to process and no time spent reconciling errors that frequently occur when filling out forms manually.

2) Trust and transparency

There is no need to question whether information has been altered for personal gain because no third party is involved, and encrypted records of transactions are shared across participants.

3) Security

Blockchain transaction records are encrypted, making them extremely difficult to hack. Furthermore, because each record on a distributed ledger is linked to the previous and subsequent records, hackers would have to change the entire chain to change a single record.

4) Savings

Smart contracts eliminate the need for intermediaries to handle transactions, as well as the time delays and fees that come with them.

Some blockchain platforms support general programming languages like C++, Java and Python, whereas other platforms, such as Ethereum, have presented support for specific languages for writing smart contracts, such as Solidity [3]. I will demonstrate Solidity and its features that complement attributes of smart contracts in Section 3 when I write my own smart contract.

Smart contracts can be used in various application scenarios: financial transactions, prediction markets, Internet of Things (IoT), supply chain and cloud storage [6]. Figure 2 depicts some of these applications; however, smart contracts can evolve to have a wide range of applications

because of their utility.

Considering the security aspect of smart contracts is critical since most contracts involve financial transactions and even minor bugs can lead to great financial loss or privacy leakage. Writing secure and bug-free smart contracts is a difficult task, as previous research indicates that a significant percentage of smart contracts already deployed on the Ethereum blockchain are vulnerable [8]. A report by L. Luu reveals that more than 45 percent are buggy [9]. There have been numerous attacks on smart contracts, each costing a significant sum of money. The DAO attack and the Parity Wallet hacks are the most frequently discussed [10]. DAO (Decentralized Autonomous Organization), an Ethereum well-known crowdfunding smart contract, was attacked in June 2016 due to a bug in its code, resulting in a 60 million USD loss [11].

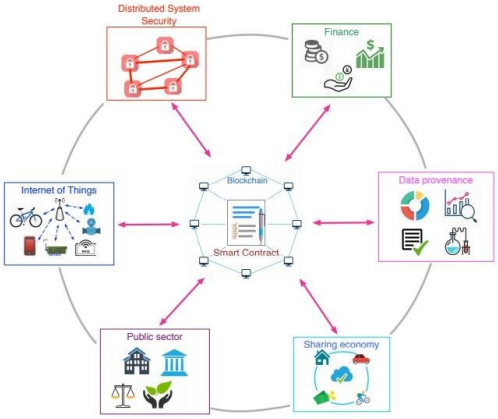

Figure 2. Common applications of smart contracts [7]

1.2. *Blockchain Overview*

Understanding how smart contract security matters, it is crucial first to relate it to the blockchain and its properties.

Blockchains are distributed tamper-resistant and cryptographically secure databases that allow transactions to be processed without the need for a trusted third party [4]. Blockchain is a chain of chronological blocks, and as shown in Figure 1, each block is identified by its hash value, which links to the previous block by referencing the previous block's hash. The only exception is the first block (termed the "Genesis block"), which does not include the previous block's hash value and can thus be considered the ancestor block [3]. Once something enters the blockchain, it can not be deleted and stays there forever. This makes it a perfect place to store property rights, credentials, identities, and agreements. The previously stated characteristic of blockchains makes them ideal platforms to deploy and execute smart contracts. We must distinguish between permissioned and permissionless blockchains. A permissioned blockchain platform, such as Fabric, limits the number of participants who can contribute to system-state consensus [4]. In other words, only a select group of authorized participants can validate transactions. To build trust and validate transactions, permissionless platforms employ consensus mechanisms such as proof-of-work or proof-of-stake. Because the permissionless blockchain is decentralized, anonymous, and equally accessible to everyone, its ability to build trust and scale is limited, resulting in low network throughput [4]. Some of the most well-known blockchain platforms used for smart contract deployment include Bitcoin, Hyperledger Fabric and Ethereum [4].

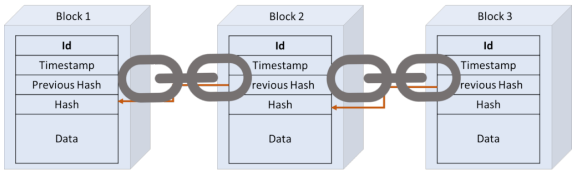

Figure 3. Blockchain hashing representation [13]

2. Writing a beginner contract

2.1. *Solidity and Ethereum*

As mentioned in section 1.1, smart contracts have a wide range of applications due to their versatility. Smart contracts are widely used in financial transactions because they have the potential to reduce financial risks, reduce administration and service costs, and improve financial service efficiency. As an example of a beginner smart contract, I created a smart contract that provides this service. To write my sample contract, I used the Solidity programming language, which runs on the Ethereum Virtual Machine. To deploy and test it, I used the Remix IDE, an Ethereum-like platform for developing, deploying, and testing Ethereum-compatible smart contracts [14]. The Ethereum Virtual Machine (EVM) is a Turing-complete decentralized machine that executes smart contract applications [3]. Ethereum has its own cryptocurrency, Ether, used to pay gas fees and compensate miners for computations performed [15]. This way, the volatile value of Ether is compensated, and proof of work is achieved [16]. The amount of computing resource required for a transaction is measured in terms of gas, used as an internal price for executing a transaction. A gas price is the amount of Ether a user pays miners to have their transactions mined and added to the blockchain: the higher the gas price, the faster the miners mine [17].

2.2. *Sample Contract Breakdown*

The sample contract "PayWorkers" is written by someone with basic knowledge of smart contracts, with little to no emphasis on security. In Section 3, I will look at parts of the contract that are at risk and secure them. The contract is a financial transaction between an employer and his or her employees. The employer/owner adds employees to the payroll by creating worker structures with according variables and using the addWorker() method. He can deposit funds into the workers' accounts using the deposit() method, which the employees can withdraw once the payment deadline has passed with the withdraw() function. A specific worker can be accessed by the index after they have been added to the workers[] array. The owner can additionally immediately pay the workers with functions payEqual() and payBonus(). The latter will only pay the workers that deserve the bonus, indicated by their deservedBonus boolean variable. The balance of the total amount of Ether in the contract can be accessed with the function balanceOf(). To calculate the payOutTime variable of each worker, I used the Epoch Time Converter, which takes in time in the 24-hour format and returns an integer [18]. The contract has been documented in order to explain how each function works. The GitHub link to the PayWorkers sample contract is included in the references [19].

---

3. Security and Vulnerabilities

It is nearly impossible to make changes to smart contracts once they have been deployed on blockchains. As a result, evaluating the correctness of smart contracts prior to formal deployment is critical. However, due to the complexity of modeling smart contracts, it is difficult to verify the correctness of smart contracts. In this section, I will classify several security issues and show how they can lead to financial loss. A study by Kalra shows that 94.6% of smart contracts are vulnerable to one or more correctness issues [12].

Furthermore, smart contract security flaws are just one of the reasons for money loss. Code optimization and efficiency heavily influence the amount of money spent on a contract. As described

in section 2.1, Ethereum uses gas to run contracts and using it optimally saves money and data. The vulnerabilities highlighted in the following section will help with both of these issues: making a contract more secure and optimizing gas spending.

After introducing each smart contract vulnerability, I will relate it to my sample contract displayed in section 2. I will identify the risks in my contract and explain how the vulnerability could result in financial or data loss. In addition, I will put in place solutions to the issues that have been discussed in each section. By implementing vulnerability checks and solutions, I will effectively audit my contract and make it more secure.

3.1. *Integer Overflow/Underflow*

When dealing with financial transactions, manipulating integers, such as computing the balance of amounts or indexing variables in an array, is critical. One of Solidity's disallowed features is the use of implicit integer extension to store the result [12]. When an arithmetic operation is completed, an integer overflow or underflow occurs due to the result exceeding the storage limit. As a result, the resulting numbers may be incorrect. Because most functions include arithmetic operations, this can lead to many overflow/underflow risks, making it one likely to appear often. A common way to initiate this issue is to provide a value greater than the maximum value. In Solidity, integers are represented as uint or uint256 because they can handle 256-bit numbers [20].

Figures 4, 5, and 6 show simple code vulnerable to integer overflow/underflow attacks [21]. Overflow and underflow can be achieved in the following example by increasing or decreasing the contract's balance. Because uint8 is used, the balance's storage limit is 28 (256). Incrementing the balance to 255 will cause it to overflow and reset the value of the balance to 0. Lowering the value to 0 causes the balance value to fall to 255.

Figure 7 depicts an integer overflow risk in my sample contract. The function payEqual() is at risk: if the value of the address passed in the function and the amount paid exceed the storage limit, the balance of that account will be reset to 0. This problem can be avoided by using a require statement in which the sum of the account balance and the payment amount is greater than the account balance, as illustrated in Figure 8.

```
pragma solidity 0.7.0;

contract ChangeBalance {
    uint8 public balance;

    function increase() public {
        balance++;

    }

    function decrease() public {
        balance--;
    }
}
```

Figure 4. Integer overflow/underflow vulnerability [21]

Figure 5. Starting balance of contract [21]

Figure 6. Balance of contract after decreasing [21]

```
function payEqual(uint totalPay) public payable onlyOwner returns(bool){
    uint size = workers.length;
    uint equalPay = totalPay/size;
    require(size > 0);
    require(totalPay > 0 && owner.balance > totalPay);
    for(uint i = 0; i < size; i++) {
        workers[i].amount += equalPay;
        workers[i].walletAddress.transfer(equalPay);
    }
return true;
}
```

Figure 7. Integer overflow vulnerability

```
function payEqual(uint totalPay) public payable onlyOwner returns(bool){
    uint size = workers.length;
    uint equalPay = totalPay/size;
    require(size > 0);
    require(totalPay > 0 && owner.balance > totalPay);
    for(uint i = 0; i < size; i++) {
        require(workers[i].amount + equalPay >= workers[i].amount);
        workers[i].amount += equalPay;
        workers[i].walletAddress.transfer(equalPay);
    }
return true;
}
```

Figure 8. Integer overflow mitigation

3.2. *Reentrancy*

Reentrancy requires the use of two smart contracts, one with a vulnerable function and the other with a call to the vulnerable function that recursively drains funds from the first [10]. Reentrancy attacks are carried out by making a call to an external malicious contract, which allows it to call the vulnerable function recursively, typically transferring funds [10]. This creates a loop that drains all of the funds from the original contract and can be disastrous if not secured. If the balance of an account in the transaction is updated or checked after the call, the function will be interrupted in the middle of the call, making a reentrancy attack possible [12].

Figures 9 and 10 depict how a malicious contract can attack a vulnerable one with a reentrancy attack. In this example, reentrancy occurs when the user is sent the requested amount of Ether, and the attacker uses the function withdraw() [22]. Even though the attacker has already received tokens, he can transfer them because his balance has not yet been reset to zero. The attack function invokes the victim's contract's withdraw() function. The fallback function calls the withdraw function when the token is received, and since the check was successful, the contract sends the token to the attacker, triggering the fallback function.

A reentrancy attack is possible in the PayWorkers contract, as shown in Figures 11 and 12. It works on the same principle as the example contract: after the withdrawAll() function gets invoked, it sends Ether to the caller using the call() function, then invoking the fallback function. Two ways to mitigate the issue, in this case, are: 1) Switch lines 179 and 180 in Figure 11 2) Create a function modifier that employs a boolean flag that is set to true before the vulnerable function is executed and resets to false immediately afterward (Figure 13).

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

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

    function withdraw() public {
        uint bal = balances[msg.sender];
        require(bal > 0);

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

        balances[msg.sender] = 0;
    }


}
```

Figure 9. Example contract with vulnerable function [22]

```
contract Attack {
    DepositFunds public depositFunds;

    constructor(address _depositFundsAddress) {
        depositFunds = DepositFunds(_depositFundsAddress);
    }

    // Fallback is called when DepositFunds sends Ether to this contract.
    fallback() external payable {
        if (address(depositFunds).balance >= 1 ether) {
            depositFunds.withdraw();
        }
    }

    function attack() external payable {
        require(msg.value >= 1 ether);
        depositFunds.deposit{value: 1 ether}();
        depositFunds.withdraw();
    }


}
```

Figure 10. Example malicious contract [22]

```
173     function withdrawAll(address payable walletAddress) public payable {
174         uint i = getWorker(walletAddress);
175         require(msg.sender == workers[i].walletAddress, "You must be a worker to withdraw!");
176         require(workers[i].withdrawable == true, "You need to wait for your pay-out time");
177         uint amountToWithdraw = getBalance[msg.sender];
178         if(amountToWithdraw > 0) {
179             msg.sender.call{value:(getBalance[msg.sender])};
180             getBalance[msg.sender] = 0;
181         }
182     }
```

Figure 11. Sample contract with vulnerable function [12]

```
contract AttackContract {
    function attack() private {
        PayWorkers attacker;
        attacker.withdrawAll();
    }
}
```

Figure 12. Potential malicious contract [12]

```
bool internal flag;

modifier preventEntry() {
    require(!flag);
    flag = true;
    _;
    flag = false;
}
```

Figure 13. Using a function modifier to prevent reentrancy attack [22]

3.3. *Unchecked send*

Solidity has three functions that allow Ether transactions: address.transfer(amount), address.call.value(amount)(), and address.send (amount) [23]. The transfer function is the safest of the three because it throws an exception when executed incorrectly [24]. That is why I made it the primary method of transferring funds in the PayWorkers contract. The call function is vulnerable to reentrancy attacks, as shown in section 3.2 and the send function is vulnerable to the unchecked send vulnerability. The unchecked send vulnerability is caused by a failure of the send() function, which can occur in one of two ways. The recipient has run out of gas, or the call stack depth has reached 1024 [25]. If the send() function fails, the recipient will not receive the funds, causing a function to be altered and money to be frozen or lost.

Figure 14 depicts an example function that is vulnerable to unchecked send. If the send() method fails, even if the boolean prizePaidOut is set to true, the winner will not actually be paid. As a result, the condition in the if statement always returns false, preventing the winner from ever being paid. Because the send() method returns a boolean, an easy solution to the vulnerability is to use an if statement to check if the send() method returns true. As shown in Figure 15, this would ensure that the winner is compensated correctly. The PayWorkers sample contract has a similar issue and solution in the function payBonus() pictured in Figures 16 and 17. The worker will get paid the bonus if the boolean deservedBonus is true when creating a worker object, the worker is able to withdraw, the year has ended, and if they have not been paid yet. They will not be eligible for the bonus again once they have received it.

```
if (gameHasEnded && !( prizePaidOut ) ) {
  winner.send(1000); // send a prize to the winner
  prizePaidOut = True;
}
```

Figure 14. Unchecked send example [25]

```
if (gameHasEnded && !( prizePaidOut ) ) {
  if (winner.send(1000))
    prizePaidOut = True;
  else throw;
}
```

Figure 15. Unchecked send example solution [25]

```
function payBonus(uint totalBonus) public payable onlyOwner {
    uint endOfYear = 1672531201;
    bool paidBonus = false;
    uint bonusAmount = totalBonus/workers.length;
    for (uint i = 0; i < workers.length; i++) {
        paidBonus = false;
        if(workers[i].payOutTime > endOfYear && workers[i].withdrawable && workers[i].deservedBonus && !(paidBonus)) {
            (workers[i].walletAddress.send(bonusAmount));
            paidBonus = true;
            workers[i].amount += bonusAmount;

        }
    }
}
```

Figure 16. Sample contract unchecked send

```
function payBonus(uint totalBonus) public payable onlyOwner {
    uint endOfYear = 1672531201;
    bool paidBonus = false;
    uint bonusAmount = totalBonus/workers.length;
    for (uint i = 0; i < workers.length; i++) {
        paidBonus = false;
        if(workers[i].payOutTime > endOfYear && workers[i].withdrawable && workers[i].deservedBonus && !(paidBonus)) {
            if(workers[i].walletAddress.send(bonusAmount)) {
                paidBonus = true;
                workers[i].amount += bonusAmount;
            }
        }
    }
}
```

Figure 17. Sample contract unchecked send solution

3.4. *Floating Pragma*

The smart contract compiler version is determined by the solidity pragma, typically written in the first line of code. The code in the smart contract should be compiled with the latest version of Solidity. The compiler will generate an error if the pragma and compiler versions do not match, as instructed by the pragma version [26]. The floating pragma can be avoided by using a caret sign(^) before the compiler version, as illustrated in Figure 18 [27]. The caret sign specifies the exact version of the compiler used to compile the smart contract (0.8.17 in the example). When the caret sign is not used, the contract can be compiled with any compiler version ranging from 0.8.0 to 0.8.17, which might result in an error. Using an outdated compiler version can be dangerous because the older version might have public vulnerabilities. Using a very recent compiler version with unresolved fixes can also lead to errors.

```
pragma solidity ^0.8.17;

// vulnerable

contract PayWorkers {
```

Figure 18. Floating pragma vulnerable example

```
pragma solidity 0.8.17;

// safe

contract PayWorkers {
```

Figure 19. Floating pragma safe example

3.5. *Access Control*

Access control regulates access to smart contract resources and allows different users to perform specific tasks [28]. Access control is critical for smart contract security because it prevents hostile users from using or accessing smart contract functions and data. Solidity includes built-in access control via a modifier, as shown in Figure 19. The onlyOwner() modifier is very useful for the PayWorkers sample contract because it can be reused for any function that should be available only to the owner. Without it, anyone could use the contract's functions addWorker(), deposit(), payEqual(), and payBonus(), which is a major security flaw. The PayWorkers smart contract will have access control and remain secure by including the onlyOwner() modifier in all of these functions. Access control for workers is achieved by using a require statement in the same manner as in Figure 19 and is needed for functions withdraw() and withdrawAll().

Another way to access control of the smart contract is with function visibility. Controlling which parties can access and view what parts of the contract is a vital part of achieving access control. The visibility of a function determines which of the three types of callers can execute it:

- the main contract

- contract inherited from the main contract

- an outside contract

The visibility keyword comes after a function's parameter list, and Solidity includes four keywords for specifying function visibility [29]:

1. "private" – can only be called by the main contract

2. "internal" – can be called by the main and the inherited contracts

3. "external" – can only be called by an outside contract

4. "public" – can be called by all

This vulnerability leans on human error and forgetfulness, as the function visibility vulnerability occurs when the smart contract writer forgets about function access control. The default visibility property of functions in Solidity is public, which can cause a slew of security issues if not correctly declared. Figure 20 illustrates an example of an overlooked function visibility in the sample contract which might lead to security breaches. Because the function balanceOf() is public, anyone can call it and see how much Ether is in the contract at any given moment. The issue will be solved by replacing the public keyword with the internal keyword, allowing only the main and inherited contracts to call the function.

```
modifier onlyOwner() {
    require(msg.sender == owner, "Only the owner is allowed to do this!");
    _;
}
```

Figure 20. Solidity access control using a modifier

```
function balanceOf() public view returns(uint) {
    return address(this).balance;
}
```

Figure 21. Improper function visibility

4. Auditing a contract

Two critical components of smart contracts to ensure they read, check, and execute conditions and rules properly are:

1) Writing smart contracts

2) Auditing smart contracts

Writing a smart contract entails all information covered in Sections 1 and 2. Auditing smart contracts means attacking our contract to check for potential risks and security issues. This way, we can get a step ahead of potential threats and understand what needs to be done to ensure the contract is secure. There are automated and semi-automated tools used to audit smart contracts such as Slither, Mythx, Mythril and Oyente [10], however this paper focuses on a manual audit of the PayWorkers contract [19]. Manual auditing is the ability to understand the logic of the code at the code level and identify logical loopholes. In Section 3, the paper analyzed 5 smart contract vulnerabilities and provided solutions to improve the contract. The resulting contract is a manually audited version of the PayWorkers contract and can be seen on GitHub using the link in reference [30].

5. Conclusion

This paper presents an overview of up-date smart contract security. In particular, I first provided a background overview of blockchain technology and its relation to smart contract security. Using this beginner information, I wrote a sample smart contract in Solidity, offering one of the most common practical applications of smart contracts. Introducing relevant information on the Ethereum Virtual Machine and smart contract security, I followed with an analysis of the following smart contract vulnerabilities:

- Integer overflow and underflow
- Reentrancy attacks
- Unchecked send
- Floating pragma
- Access control

I then introduced solutions to these issues and explained their functionality through implementation on my sample contract. To conclude, I presented my audited smart contract to illustrate how the solutions to the presented issues improved the security of my contract.

References

[1] S. Nick. (2017). The Idea of Smart Contracts. [Online]. Available: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/ Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html

[2] S. Nick. (1997). Formalizing and Securing Relationship on Public Networks. [Online]. Available: http://rstmonday.org/ojs/ index.php/fm/article/view/548/469

[3] Rouhani, Sara, and Ralph Deters. "Security, performance, and applications of smart contracts: A systematic survey." IEEE Access 7 (2019): 50759-50779.

[4] Yaga, Dylan, et al. "Blockchain technology overview." arXiv preprint arXiv:1906.11078 (2019).

[5] Nzuva, Silas. "Smart contracts implementation, applications, benefits, and limitations." School of Computing and Information Technology, Jomo Kenyatta University of Agriculture and Technology, Nairobi, Kenya (2019).

[6] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin and F. -Y. Wang, "An Overview of Smart Contract: Architecture, Applications, and Future Trends," 2018 IEEE Intelligent Vehicles Symposium (IV), 2018, pp. 108-113, doi: 10.1109/IVS.2018.8500488.

[7] "Applications of smart contract 3.1. finance: Smart contracts can ..." [Online]. Available: https://www.researchgate.net/figure/Applications-of-Smart-Contract-31-Finance-Smart-contracts-can-potentially-reduce_fig2_340376424.

[8] 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 Financial Cryptography and Data Security. New York, NY, USA: Springer, 2016, pp. 7994.

[9] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in Proc. ACMSIGSAC Conf. Comput. Commun. Security, Oct. 2016, pp. 254269.

[10] S. Sayeed, H. Marco-Gisbert and T. Caira, "Smart Contract: Attacks and Protections," in IEEE Access, vol. 8, pp. 24416-24427, 2020, doi: 10.1109/ACCESS.2020.2970495.

[11] V. Buterin. (2016). Critical Update Re: Dao Vulnerability Ethereum Blog. [Online]. Available: https://blog.ethereum.org/2016/06/17/criticalupdate- re-dao-vulnerability/

[12] Kalra, Sukrit, et al. "Zeus: analyzing safety of smart contracts." Ndss. 2018.

[13] “Blockchain Explained.” ShootSkill, 15 May 2021, https://shootskill.com/blockchain/blockchain-explained/.

[14] “Ethereum IDE & Community,” Remix. [Online]. Available: https://remix-project.org/.

[15] “What is ether (eth)?,” ethereum.org. [Online]. Available: https://ethereum.org/en/eth/.

[16] Vujičić, Dejan, Dijana Jagodić, and Siniša Ranđić. "Blockchain technology, bitcoin, and Ethereum: A brief overview." 2018 17th international symposium infoteh-jahorina (infoteh). IEEE, 2018.

[17] Zheng, Zibin, et al. "An overview on smart contracts: Challenges, advances and platforms." Future Generation Computer Systems 105 (2020): 475-491.

[18] “Epoch converter,” Epoch Converter - Unix Timestamp Converter. [Online]. Available: https://www.epochconverter.com/.

[19] (n.d.). PayWorkersVulnerable. GitHub. https://gist.github.com/jatadi/b0b70d41aaa5d31d139f648134912289

[20] “Types,” Types - Solidity 0.8.17 documentation. [Online]. Available: https://docs.soliditylang.org/en/v0.8.17/types.html.

[21] K. Polak, “Hack solidity: Integer overflow and underflow,” HackerNoon, 17-Jan-2022. [Online]. Available: https://hackernoon.com/hack-solidity-integer-overflow-and-underflow.

[22] K. Polak, “Hack solidity: Reentrancy attack,” HackerNoon, 17-Jan-2022. [Online]. Available: https://hackernoon.com/hack-solidity-reentrancy-attack.

[23] Z. Mohammed, “Solidity-transfer vs send vs call function,” Medium, 25-Jun-2022. [Online]. Available: https://medium.com/coinmonks/solidity-transfer-vs-send-vs-call-function-64c92cfc878a.

[24] K. Bulgakov, “Three methods to send ether by means of solidity,” Medium, 01-Jul-2018. [Online]. Available: https://medium.com/daox/three-methods-to-transfer-funds-in-ethereum-by-means-of-solidity-5719944ed6e9.

[25] “Scanning live ethereum contracts for the ‘unchecked-send’ bug,” Hacking Distributed. [Online]. Available: https://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/.

[26] ImmuneBytes, “Floating pragma: The ultimate guide,” ImmuneBytes, 17-Nov-2022. [Online]. Available: https://www.immunebytes.com/blog/floating-pragma/.

[27] J. Chittoda, “Mastering blockchain programming with solidity,” O’Reilly Online Learning. [Online]. Available: https://www.oreilly.com/library/view/mastering-blockchain-programming/9781839218262/d1250994-b952-4d5e-9cde-1b852c18b55f.xhtml.

[28] B. Liu, S. Sun and P. Szalachowski, "SMACS: Smart Contract Access Control Service," 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2020, pp. 221-232, doi: 10.1109/DSN48063.2020.00039.

[29] Ronnie Payne , “How to manage visibility of variables and functions in solidity,” Developer.com, 02-Jun-2022. [Online]. Available: https://www.developer.com/languages/variable-function-visibility-solidity/.

[30] (n.d.). PayWorkersSafe. GitHub. https://gist.github.com/jatadi/6fd24e320b3d6bc143f940789bcfdd1b

6. Instructions

Remix IDE was used to write, deploy and execute the PayWorkers contracts. The code should be copied into the IDE in a new contract which can be created as illustrated in Figure 22. Use the compiler icon to compile the code. Click on the "Deploy and run transactions" icon to deploy the contract and test its functionality (Figure 23). To show and use the functions of the contract, click on the arrow in the bottom left corner under the "Deployed Contracts" tab (Figure 24). The addresses of workers are represented by the addresses under the account tab (Figure 25) and should be used as arguments in the functions. Examples of using some of the functions are pictured in Figures 26 – 28.

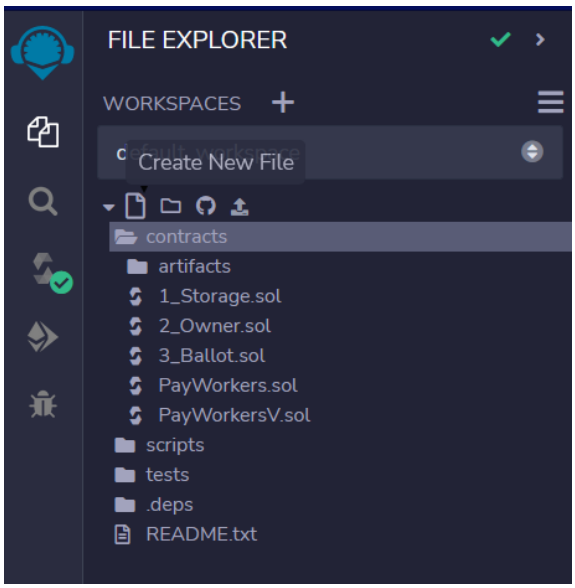

Figure 22. Creating a new contract in Remix

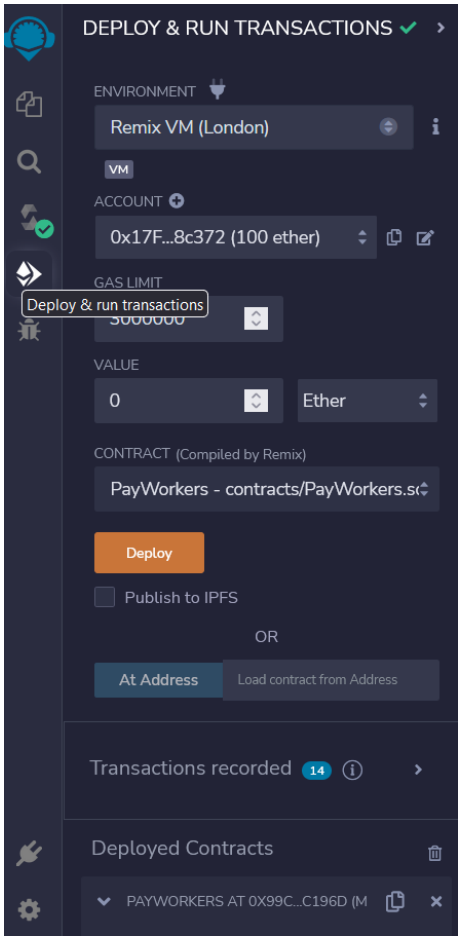

Figure 23. Deploy and run transactions

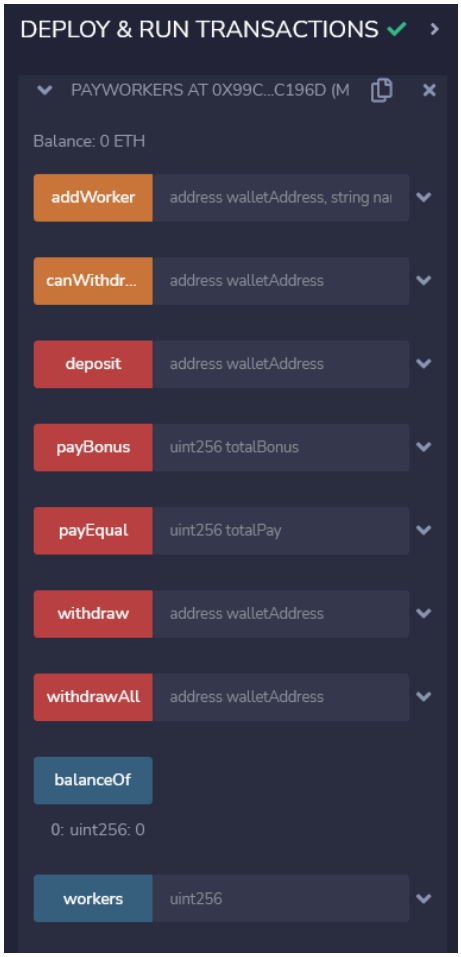

Figure 24. Functions from the PayWorkers contract

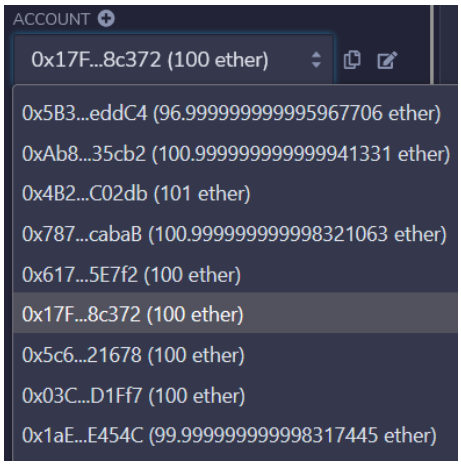

Figure 25. Addresses of accounts

Figure 26. Getting the worker at index 0

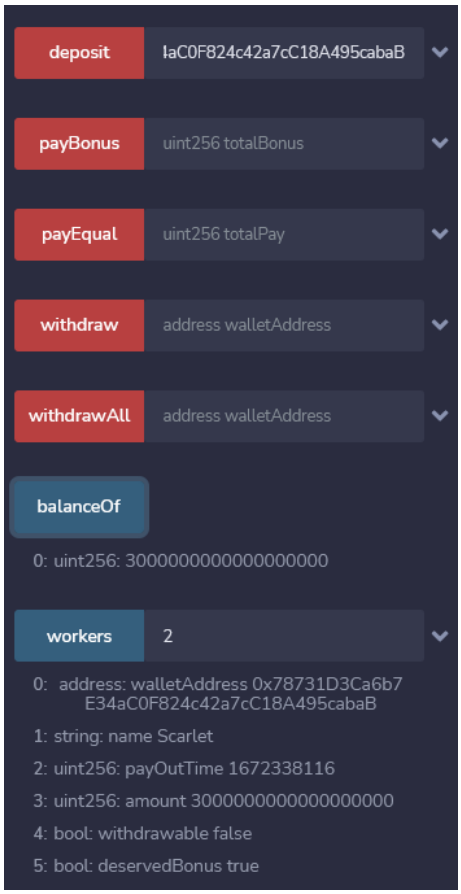

Figure 27. Depositing 3 Ether to worker at index 2

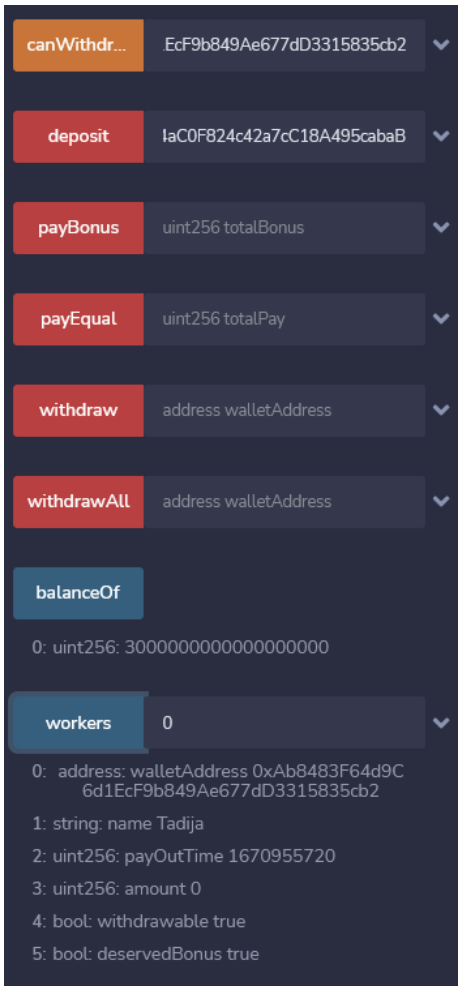

Figure 28. Checking if worker at index 0 can withdraw after payOutTime passed

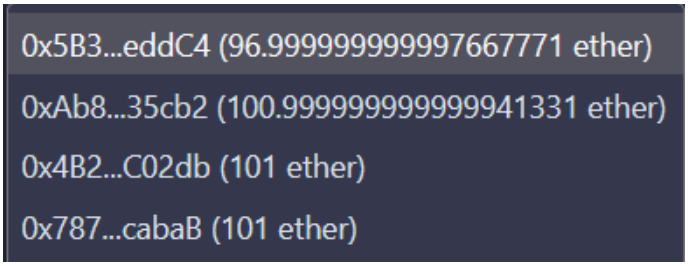

Figure 29. Balances in the accounts of the employer and 3 workers after the payEqual(3 Ether) function was called