Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

A Survey of Vulnerabilities on Ethereum Smart Contracts (SoK)

NIANDI YANG[1]
[1]Department of Computing, The Hong Kong Polytechnic University

ABSTRACT Smart contracts are self-executing programs stored on the blockchain that can process transactions without third-party trust. Due to its decentralized, self-executing, and automated nature, it has become a new trending technology in many application areas such as DeFi, supply chain management, NFT, DAO, etc. The market size of smart contracts already grew to 315.1 million USD in 2021. At the same time, the vast volume of transactions makes smart contracts easily prone to malicious attacks. Rouge trader stole $117 million from DeFi platform Mango by exploiting the loophole of the smart contract on October 12, 2022. Therefore, it is necessary to provide a comprehensive summary of the vulnerabilities that often occur in smart contracts and the measures to address them. In this paper, common vulnerabilities in smart contracts and their solutions are summarized.

Contents

I. INTRODUCTION

In 1994, Nick Szabo introduced the concept of smart contracts, defined as "computerized transaction agreements that enforce the terms of a contract." Limited by state of the art at the time, it was not developed into a specific technology but remained only an idea. However, with Satoshi Nakamoto's invention of Bitcoin in 2008 [1], blockchain technology has gained more and more traction in recent years, making it possible to implement smart contracts. Bitcoin allows for smart contracts, but opcode programming is necessary. This limits the functionality and potential of Bitcoin smart contracts. In 2015, the public blockchain Ethereum was launched, which marked a significant turning point for smart contracts. Ethereum smart contract refers to immutable programs that run deterministically on a decentralized Ethereum World computer as part of the Ethereum network protocol [1]. This technology brings blockchain 2.0, which allows users not only able to exchange cryptocurrency but also to perform sophisticated tasks.

Decentralized applications (DAPPs) are one of the essential parts of blockchain 2.0, built on a decentralized network composed of smart contracts and a front-end interface. Smart contracts are used explicitly as APIs that connect DAPPs with blockchain. When no trusted institution holds assets or data on both sides of a transaction, DAPPs are often used instead of traditional applications. Decentralized Finance (DeFi) is the most popular application for smart contracts. DeFi DAPPs provide decentralized peer-to-peer

financial services to challenging centralized financial systems. Various services are now available in DeFi, such as exchanging cryptocurrencies, donations, investments, and loans. Using new technologies like cryptocurrencies and smart contracts, DeFi will change how financial services are provided by connecting borrowers, lenders, and developers. The expectation for the future is that DeFi will provide financial opportunity to all and bring fee-free transactions, stable exchange rates, and innovative solutions to other issues. Smart contracts also show potential in other areas, such as trustless e-voting, IoT, patent issuance and tracking, supply chain management, identity verification, etc.

Ethereum replaces Bitcoin's more restrictive language with Solidity, a high-level language with Turing completeness. As a result, developers can handle any task with a more extensive instruction set, not just cryptocurrency transactions. While this has obvious benefits, it also brings vulnerabilities, many of which have been proven to be related to Solidity's characteristics. In addition to the problems caused by Solidity, the way EVM handles state and calls external calls can lead to its attack. When smart contracts provide transparency, it also exposes the state variable of code to everyone. The bad actor can modify the state variable or create a malicious contract to perform an attack. By exploiting the vulnerabilities in smart contracts, attackers have already stolen millions of dollars from Ethereum. Hence, ensuring the security of smart contracts becomes the most critical issue. This paper focuses on the vulnerabilities of smart contracts and discusses solutions for each of them.

II. SOLIDITY-RELATED VULNERABILITIES

Although Solidity is the most functional language, it is also the least secure [2]. Some common vulnerabilities of Solidity are listed below.

A. INTEGER OVERFLOW AND UNDERFLOW

This vulnerability is a common attack on smart contracts before Solidity 0.8. It occurs when the value provided is below the lower bound (underflow) or greater than the upper bound (overflow). In Solidity, if the return value overflows, it does not throw an exception but takes the value of the overflow value modulo the integer range. For instance, if we use uint256, $2^{256} + 3 - 1 = 2$. And when underflow occurs (the value is below the lower bound), the value will be restored to the maximum value instead of 0. A typical underflow attack is the Proof of Weak Hands Coin (PoWH Coin) [3]. One of PoWH Coin's mechanisms is that the coins sold from the first account will be deducted from the balance of the second account. This logic leads to an underflow that resets the balance of PoWH coins in the second account to 2^{256}-1, which results in the attacker gaining illegal access to 2,000 ETH. Luckily, the Solidity 0.8 compiler will automatically check the overflow and underflow. When these two situations occur on operation on integers, it reverts to the initial state. For this reason, it is recommended that everyone should use Solidity version 0.8 and above.

B. UNINITIALIZED STORAGE POINTERS

All uninitialized struct/mapping/array local variables will be mapped to slot[0], so some variables may be overwritten if there are multiple complex data type variables. This can lead to unexpected behavior used to perform honeypot attacks. The OpenAddressLottery [4] and CryptoRoulette [5] honeypots are attacks based on this vulnerability. In Solidity 0.5 and above, programmers must explicitly declare storage and memory keywords, which solves this problem.

C. FAKE AUTHENTICATION OF TX.ORIGIN

This attack is a classical phishing attack to contract [6]. The global variable tx in Solidity represents the transaction, and tx.origin represents the original EOA address that initialized this transaction. It is often confused with the msg.sender, which means the direct sender of the transaction. When a contract uses tx.origin to authenticate the owner's identity, it might be attacked because these two values can differ. This simple programming mistake can be avoided using msg.sender to identify the owner. If a contract must use tx.origin, it is recommended to add the require statement: require(tx.origin==msg.sender); to ensure security.

D. CALL AND DELEGATECALL TO UNTRUSTED CONTRACTS

Call and delegatecall are low-level functions that call other contracts. They are the same, except that delegatecall is executed in the context of the calling contract, and msg.sender and msg.value are not changed. The mechanism of call and delegatecall can be illustrated in the following pictures.

Multiple attacks are related to Solidity's external calls. Here, we discuss the unexpected behavior caused by call and delegatecall and leave the other attacks in the following sections. For call, the first case is that if we use it to invoke an untrusted contract many times, one function can return different results since the state of the untrusted contract might change [8]. The second case is that the fallback function will be triggered when the signature function does not exist in the invoking contract or some parameters of the function are incorrectly entered [9]. For delegatecall, it runs in the context of our contract, allowing the calling contract to have full access to our contract's state variables and manipulate it maliciously. One possible solution is to avoid external calls as much as possible, especially the delegatecall, unless it has the library keyword. More specifically, the programmer can add a modifier to prevent delegatecall. In the case of the Parity wallet, initWallet was not checked to prevent an attacker from calling initMultiowned after the contract was initialized. This vulnerability allowed a hacker to make himself the new owner of multiple Parity wallets via a library function and then call the transfer function to move the money away—this delegatecall vulnerability cost Parity over 150,000 ETH [10].

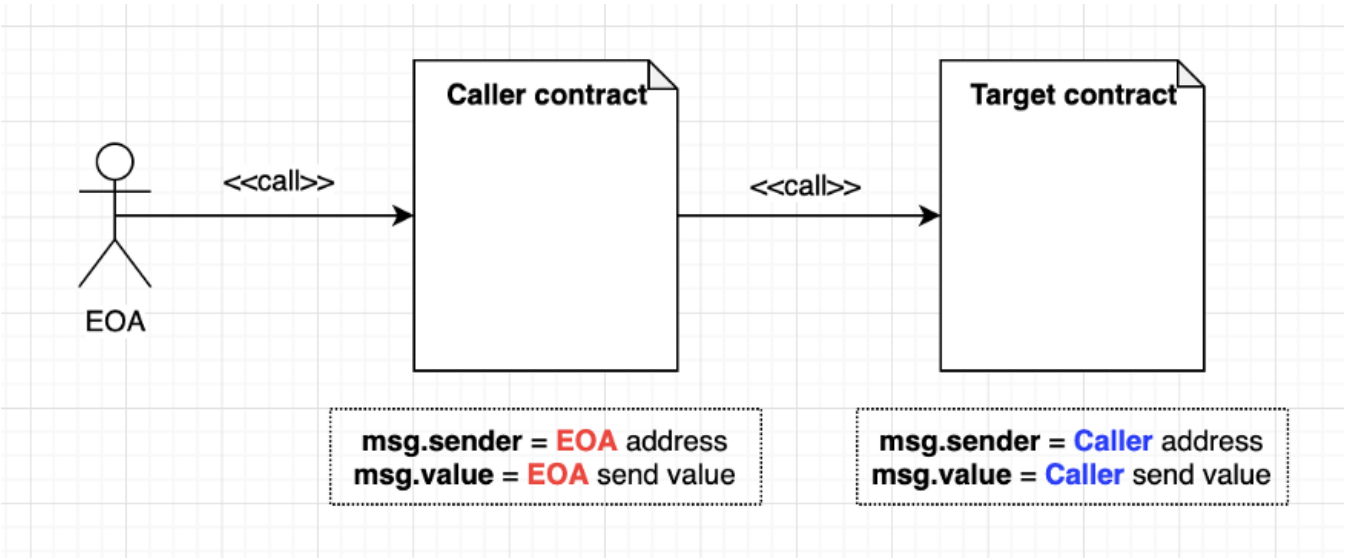

FIGURE 1. **An example of call.** [7]

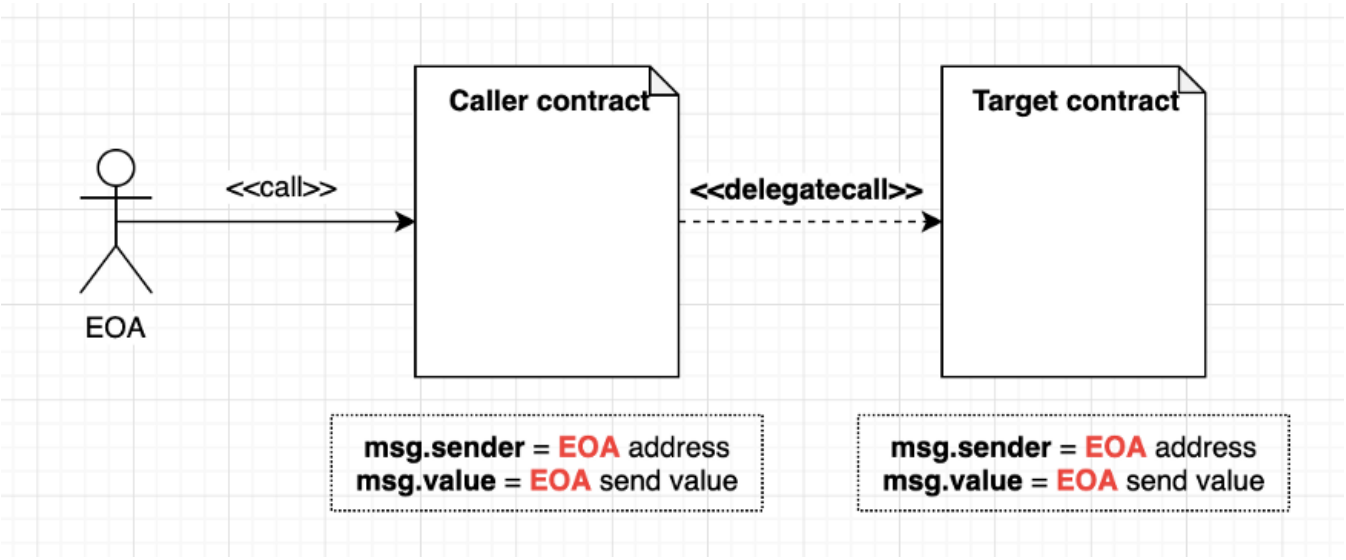

FIGURE 2. **An example of delegatecall.** [7]

E. SOLIDITY-RELATED DOS ATTACKS

DoS attacks on the blockchain can be divided into four kinds: DoS with unexpected revert, DoS with unbounded operations, and DoS with block stuffing [11], and DoS with 51% attack. This section only talks about the previous two Solidity-related DoS attacks, leaving the DoS with block stuffing and DoS with 51% attack behind.

1) DoS with unexpected revert

This attack allows a transaction to be interrupted and not continue forever or for a long time. There are two classic cases: bid contract [12] and refund contract [13]. In the bid contract, the attacker can win any auction. Once the refund to the previous leader fails, it will revert to its previous state. This means that a malicious bidder can win the auction while all refunds to their address fail. In this case, other bidders cannot invoke the bid function anymore. The refund contract fails in a different mechanism. There exists a function that refunds to all EOA addresses by traversing an array. Whenever a refund fails, the EVM reverts, which means that this iteration of the refund is equivalent to a while(true) loop and never ends. One solution to this attack is the pull over push payment that guarantees the atomicity of transactions and programs. This payment method avoids combining multiple Ether transfers into a single transaction by allowing users to withdraw their own funds instead of

pushing funds to them automatically.

2) DoS with unbounded operations

As the name implies, this attack is caused by traversing an array of unknown size. Without knowing the size, it is impossible to know the bounds of the consumed gas. The refund contract we discussed in DoS with unexpected revert is also a case of this attack. It is reminded that each block's "gas limit" field specifies the maximum amount of gas a transaction within the block can consume. This vulnerability occurs when the amount of gas needed to execute a contract exceeds the block's gas limit. This attack is also a programming mistake that can be avoided. As for the solution referred to [1], dynamically sized arrays should be avoided when writing contracts. When a long list of values is expected, it is always recommended trying to use mapping instead of arrays. It also requires caution when interacting with arrays of unknown size that exist in external contracts. When a contract needs to traverse an array, it should keep track of the loop. If the traversal fails in the middle, the sender of the transaction gets the information from the previous breakpoint when the contract is re-invoked and continues execution at that point [11].

F. REENTRANCY

It is worth noting that when your contract calls a function of an external contract, the external contract can also call your function, even recursively. Reentrancy attacks can be interpreted as an attacker repeatedly invoking state variables when they have not been updated correctly and are shared in another function or contract. There are three types of reentrancy attacks: single-function reentrancy attack, cross-function reentrancy attack, and cross-contract reentrancy attack. Single-function reentrancy attack is the most common reentrancy attack for smart contracts. A classic case is that the contract uses the balance variable in the withdraw function shown in figure 3.

An attacker can keep calling the withdraw function and drain the fund of the contract when it fails to update its status. In this single-function reentrancy attack sample, after transferring the correct amount of money msg.sender.call.value(amount)(), the attacker's fallback function calls withdraw again and transfers more money before balances[msg.sender] = 0 can prevent further transfers. The attacker will continue to call the withdrawal function until there is no ether left or the limit of the stack is reached, or the gas is fully consumed at the time of execution. In 2016, DAO was compromised by such an attack, with the attackers stealing 3.6 million ETH.

The remaining two attacks are more difficult to detect than single-function reentrancy attacks. Cross-reentrancy attacks can easily occur when multiple functions share the same state variables, and some update insecurely. Cross-contract reentrancy is like cross-function reentrancy except that the state variables are shared between two contracts.

These three attacks can be resolved using the checks-effects-interactions pattern or mutually exclusive lock. The checks-effects-interactions pattern implies that the state variables should already be updated correctly before contact with other contracts. In figure 3, we can exchange the order of balances[msg.sender]=0 and msg.sender.call.value(amount)() to fit this pattern for preventing reentrancy attack. Mutual exclusion locks are another solution to the reentrancy attack, aiming to lock the state of a contract so that only the lock owner can modify the state. Since functions in the same contract are not repeatedly called, the reentrancy attack is solved. This method is more effective for cross-function reentrancy attacks. A widely used way to implement locks is to use OpenZeppelin's ReentrancyGuard contract module and the nonReentrant modifier. However, the lock must be implemented correctly to avoid the case of an attacker locking your contract and preventing it from being released, rendering your contract invalid. Furthermore, how to balance the extra performance costs of locks and the security of contracts is also a question.

It is worth noting that strategies to prevent reentrancy attacks are not set in stone. Some solutions may become invalid. Until the Constantinople update was delayed, it was thought that send and transfer were better than call. The gas limit for these two functions is 2300 units, which is insufficient for recursive calls by reentrancy attackers. However, EIP 1884 shows that gas costs can and will change [14], so we cannot rely on the gas limit of these two functions anymore. This makes the call function following the checks-effects-interactions pattern a much more secure choice.

In summary, there are some recommendations for the prevention of reentrancy. First, do not make external calls if you are not sure the contract being called can be trusted. If your contract must make external calls, the checks-effects-interactions pattern should always be followed to provide security. Second, you should double-check and try to update all state variables when they are shared, which can significantly improve your contract's resistance to reentrancy attacks.

G. SELFDESTRUCT ATTACK

The function selfdestruct(address) removes all bytecodes from the contract address and sends all stored ethers to the specified address. Nonetheless, there is a particular case where the fallback function will not be executed if the receiving address is a contract. That means an attacker can use the selfdestruct() function to create a contract, send it Ether, call selfdestruct(target) and force the Ether to the target. This vulnerability is due to a flaw in this.balance. If a contract function has a conditional statement that determines whether this.balance is below a certain amount, then that statement might be bypassed. One solution is that the contract should not rely on this.balance, as it can be easily manipulated. If you need the exact value of the Ether deposited, you should use a custom variable that is incremented to keep track of

```
function withdraw() external {
    uint amount = balances[msg.sender];
    require(msg.sender.call.value(amount)());
    balances[msg.sender] = 0;
}
```

FIGURE 3. An example of the reentrancy attack on the withdraw function.

the Ether deposited. This prevents your contract from being affected by the forced Ether sent via selfdestruct().

H. TYPE CASTS

It is reminded that contract written in Solidity can invoke another contract by directly referencing an instance of the invoked contract. When the caller declares the interface of the callee and forces a conversion to the address of the callee when the call is executed. Performing certain type checks may mislead the programmer into believing that all type checks have been performed. If the called function does not exist in the called contract, then the called contract fallback will not throw any exceptions to alert the caller. Likewise, if a function with the same signature is called or a call is made to something other than a contract address, there are no exceptions thrown [9]. This vulnerability is caused by Solidity's unreliable data type system. Currently, there is no feasible way to resolve it.

I. OTHER VULNERABILITIES

Other vulnerabilities are function default visibility, floating pragma, and multiple inheritances [8]. In Solidity, functions are public by default. This means that any malicious EOA address or contract can call your contract's functions and execute its code. The DAO attacks mentioned before are one of the victims of function default visibility. Fortunately, the visibility modifier must be added in subsequent versions of Solidity. With floating prgama, developers will not know which version the compiler is using to compile your code, which means that unexpected behavior may occur that cannot be located. Therefore, it is recommended to choose a fixed version of Solidity to find bugs. Multiple inheritance is supported by Solidity, which introduces an ambiguity known as the "diamond problem." If multiple parent contracts implement the same functionality, Solidity uses C3 linearization to determine which parent contract takes precedence. If developers are not aware of this, it can result in unexpected behavior. One solution to this problem is that parent contracts should be added according to the C3 linearization workings, from the most generic to the most specific contracts.

III. EVM AND BLOCKCHAIN VULNERABILITIES

In addition to vulnerabilities caused by Solidity features, there are also some vulnerabilities due to EVM features and blockchain features.

A. CALL STACK SIZE LIMIT

This attack is an EVM attack, caused by its imperfect execution mode [11]. Each time a contract invoke itself or other contract, the call-stack depth will increase by one. When it reaches the limit of 1024, EVM will revert. An attacker could exploit this vulnerability by following a flow: generates a call stack close to 1024 depth (through a series of nested calls) and then calls a function of the victim contract that will fail on further calls. Because Solidity does not propagate exceptions in low-level external calls, if the victim contract does not handle this situation correctly, the victim contract may not notice that the call has failed. The attacker can then manage to carry out the attack successfully. This problem is now solved by the hard-fork EIP-150 of Ethereum, as the maximum forwarding gas is changed to 63/64 of the total available gas. In this case, the block gas limit needs to reach 1e8 ((63/64)^1024) to reach a depth of 1024, which means that it is impossible to reach a call stack depth of 1024.

B. SHORT ADDRESS ATTACK

This attack is an EVM attack, directly related to the parameters that EVM can accept to be padded incorrectly [15]. Specifically, a short address attack occurs when a contract receives fewer bytes than expected. EVM fills in the gaps with zeros. The deployed smart contract cannot prevent this and interprets these extra zeros as part of the correct value, leaving significant vulnerability to attackers. To solve this issue, external applications should validate all input parameters before sending them to the blockchain. In addition, it should be noted that parameter ordering plays a crucial role here. Since padding occurs only at the end, parameter ordering in smart contracts can mitigate certain forms of this attack. Another solution to this vulnerability is to check the msg.data.length and ensure the amount of data is the same as the amount expected by the contract.

C. 51% ATTACK AND ITS DERIVED ATTACKS

The 51% attack is an attack on the blockchain in which a group controls more than 50% of the network's hashing power (the calculation that solves the cryptographic hash puzzle). The group then introduces an altered blockchain to the network at a particular point in the blockchain. This is

theoretically accepted by the network, as having 51% of the nodes on the network gives the controlling party the right to change the blockchain. The attackers could prevent new transactions from being confirmed, allowing them to stop payments between some or all users. They would also be able to undo transactions completed during their control. The most severe problem is that they can double-spend cryptocurrency, which caused double spending attack.

Specifically, a double spending attack is when a coin is used multiple times. A malicious miner controlling 51% of the hash power can decide the transaction consensus and manage the currency reward, thus modifying the transaction information in the blockchain and allowing the modified block to enter the blockchain. If this happens, the attacker who initiated the modification can retrieve the used coins. Apart from double spending attack, 51% attack can also cause DoS attack [16]. If an attacker control most of the hashing power in the blockchain network, a denial-of-service attack can be sustained against a specific address or set of addresses. The attack is carried out through the following process: several transactions to be attacked are selected, and if they are contained in a block mined by another miner, the attacker can intentionally fork and remine the block for the purpose of excluding a specific transaction.

The point to emphasize here is that any attack is designed to be profitable. For large cryptocurrency systems, the 51% attack and its resulting double-spend attack is computationally nearly infeasible. And the cost of a large amount of hashing power in the cryptocurrency system outweighs the profit that an attack can bring. Potential attackers of large cryptocurrency systems prefer to mine blocks because it would be more profitable than executing a 51% attack. Ether and Bitcoin have never been successfully attacked in this way. However, for cryptocurrency systems that do not have high hashing power, it is necessary to guard against 51% attacks. One such example is the second 51% attack on Ethereum Classic in 2020 that resulted in a massive reorganization of 4,236 blocks and a successful double spend of $1.68 million worth of cryptocurrency [17]. One solution to the 51% attack and its double spending is replacing Proof of Work (PoW) with other consensus mechanisms like Proof of stake (PoS). PoS has proven to be more robust to 51% attacks than PoW due to economic defense methods [18]. Using the PoS consensus mechanism, when a malicious verifier performs the 51% attack, the normal verifier can maintain the honest chain or forcibly remove the attacker from the network and destroy their staked ETH.

D. REPLAY ATTACK

In the context of blockchain, replay attacks refer to maliciously conducting and repeating transactions on the same or another blockchain. This attack usually occurs when the blockchain is hard forked, i.e., when it changes or upgrades its protocol [19]. For Ethereum, exchanges did not discover when the Ethereum hard forked that when users withdrew ETH from the platform, they could receive the same amount of ETC. Many took advantage of this mistake by repeatedly accessing ETH on exchanges to get additional ETC [20]. After merging the Ether PoW and Ether PoS chains, the problem of replay attacks still exists if one wants to operate both chains at the same time. Since two chains have the same chainID, we can use separate wallets for networks to avoid replay attacks [21]. Some general methods also exist to prevent replay attacks, such as random session keys, attaching timestamps to messages, one-time passwords, and marking blockchain [19].

E. FRONT RUNNING

This attack is a blockchain attack. To understand the front running attack, we must first understand how transactions are processed in the blockchain. In a blockchain, transactions are generated as follows: they are gathered into blocks and incorporated into those blocks. Before transactions can be added to blocks, all nodes in the blockchain network must have knowledge of them. Due to the decentralized nature of the block creation process, the entire blockchain network must reach consensus on the transaction. When a blockchain user generates a transaction, that transaction is broadcast to the entire blockchain network. Every node that receives a broadcasted transaction adds it to the pool of unspent transactions. When constructing a new block, the block creator draws from the pool of unspent transactions. Typically, the order in which transactions are added to a block is determined by transaction fees. Even though there is typically a minimum fee per block, users can set their own fees. This means that users can pay for priority and consequently incur higher transaction fees. To maximize their profits, block creators who receive these fees will add transactions to the block based on the fees rather than the order in which they were received.

The front running attack exploits users who submit transactions but are still waiting for them to be included in the blockchain. The attacker constantly monitors the transactions sent to the memory pool and pre-mimics the results of these transactions before they are executed. This information is exploited by running his crafted transactions first, incorporating them into the blockchain, and reacting in advance to make a profit when certain conditions are met. The attacker can do this because all transactions have brief visibility in the memory pool before being executed, and therefore can see the purchase order transactions and broadcast and complete a second order before including the first one. This is known as the "front running attack." It is not easy to prevent, depending on the specific contract. In this attack, a malicious node can accept its transactions at any price. This vulnerability is caused by the state of the contract, which depends on how the node selects transactions in the block.

From the attacker's perspective, three types can launch such attacks: miners, full nodes, and super nodes. Miners conduct most attacks. Miners can manipulate the execu-

tion order of transactions and combine them with delayed transactions without broadcasting them. Full nodes can view transactions that have yet to be confirmed. Any full-node client of Ethereum can advance pending transactions by sending its adjusted transactions at a higher gas price. Super nodes in the network may influence the propagation of transactions. They can conduct attacks by manipulating the order in which miners receive transactions or preventing miners from receiving transactions. From the standpoint of attack categories, they can be further classified as shifting, insertion, and suppression. A displacement attack occurs when an attacker replaces an innocent user by maintaining a higher gasoline fee in a higher transaction order. In an insertion attack, the attacker must have the user's transaction occur after their own. For example, a user offers a higher gas price for his transaction before other users. The attacker will now trade at a lower gas cost than the user's bid, followed by a quick trade to profit from the difference. Suppression attacks, also known as "block filling attacks," typically occur when an attacker uses a high gas cost and gas volume to delay a user's transaction by consuming all the gas and filling the block's gas limit.

The solution to front running attack can be divided into the following three main aspects: transaction order, confidentiality, and design issues [22]. In the transaction order aspect, the blockchain eliminates the ability of miners to arbitrarily order transactions and attempts to make transactions obey a specific order. Three main approaches for this are First-in-first-out (FIFO) sequences, pseudorandom sequences, and forced transaction execution sequences. However, each of these methods has its own drawbacks. Transactions following FIFO order sequences might need a centralized assigner to assign sequence numbers, which decreases the decentralization characteristics significantly. For pseudorandom solution, although it can make front running statistically difficult, the effect of the solution is very small and may even be counterproductive to promote the attack. Forced transaction execution sequences mean that only one set of concurrent transactions can be confirmed, which imposes additional performance costs.

Confidentiality assumes that we can build a confidential DAPP that will not allow the front running attack because the attackers will not know the details of the transactions they want to manipulate [22]. The leading solution of this problem are commit/reveal scheme and its variant solutions. A commit/reveal scheme is an encryption algorithm that enables an individual to commit to a value while concealing it from others and retaining the ability to reveal it later. The values in a commitment scheme are non-negotiable, which means that once they have been submitted, they cannot be altered. The scheme consists of two phases: a submission phase in which values are chosen and specified and a reveal phase in which values are revealed and verified. In Ethereum, the two phases are commitment transaction and reveal transaction. The whole process can be illustrated as follows: User send commitment transaction to DAPP by Ethereum network first, when the commitment period ends, user reveals the transaction information to DAPP. One variant of commit/reveal is the submarine commitments, which extends the confidentiality of commitments and disclosures by making a commitment transaction equivalent to a transaction with a newly generated Ethereum address. This approach obscures the names of the functions and variables being called, making the DAPP safer. However, the apparent disadvantage of this strategy is that it increases transaction processing time and overhead.

The last approach is from a design perspective and aims to make DAPPs naturally robust to the preceding attacks. There is an assumption that front running is not preventable and thus a normal DAPP should be robust to front running attacks. This approach is the best and most general. If the DAPP can eliminate the benefit of malicious actors getting a head start on the application, then front running attacks can be naturally avoided [23].

F. TIMESTAMP DEPENDENCE

This attack is a blockchain attack. The timestamp dependency vulnerability occurs when a smart contract uses a node-generated timestamp value (block.timestamp) to compare with a time value that will occur in less than 900 seconds [24]. An example of this attack is smart contract roulette. The contract allows the user to get all the money by sending 10 Ether coins to the contract at the right time. The time is correct when the timestamp value retrieved from the "now" command at the time the transaction is sent should be divisible by 15. An attacker can still make the timestamp valid by increasing the timestamp presented by the node by a factor of up to 15, as the value is still less than 900. A malicious node can exploit this vulnerability and take all the funds in the contract. There are three ways to prevent timestamp dependency vulnerabilities: do not use block.timestamp value as an access control check, allow an error margin of 900 seconds, and the 15-second rule. The last 15-second rule is that if the size of your time-dependent events can vary by 15 seconds and maintain integrity, it is safe to use block.timestamp [25]. Since blockchain networks are asynchronous, each node may have a different local time, and message transmission time is not bounded so this vulnerability can happen on any blockchain.

G. DOS WITH BLOCK STUFFING

This attack is a blockchain attack. In this attack, the attacker submits a series of transactions that intentionally fill the block's gas limit so that other transactions cannot be included in the blockchain. The attacker can pay a higher transaction fee to ensure that miners process their transactions. By controlling the amount of gas consumed by the exchange, the attacker can influence the number of transactions mined and included in the block. Fomo3D came under this kind of attack in 2018 [26], the attacker only spent about 1.7 ETH to win 10,469 ETH. The attacker used many jumps command, push and dup operations to significantly

increase gas consumption. Thus, using block stuffing to win the first round of the gambling game: he ended the round by being the last buyer in block 6191896 and then put the transaction submitted to Fomo3D on hold for about 180 seconds until block 6191909. To prevent such attacks from occurring, it is important to carefully consider whether it is safe to include time-based operations in the application. In addition, using a push model instead of a pull model for payment is a better solution.

H. RANDOMNESS

This is a blockchain attack that occurs when on-chain data (e.g. blockhash, block.number, block.timestamp etc.) is used as the source of randomness [27]. It is worth emphasizing that transactions on the Ethereum blockchain are deterministic, so there is no source of randomness in Ethereum. When contracts use block variables as a source of randomness, since these sources of randomness are predictable to some extent, a malicious user can usually replicate it and rely on its unpredictability to attack the feature. The best way to address this vulnerability is never to use on-chain data as the randomness source. Another solution is using a verifiable random function (VRF) that ensures that the results are not tampered with or manipulated. Alternatively, a two-transaction system can ensure randomness when the selected block is already mined [28].

IV. CONCLUSION

The previous sections summarize common vulnerabilities in Ethereum smart contracts from Solidity, EVM, and blockchain. These are known vulnerabilities that we have discovered owing to financial losses. New vulnerabilities are likely to keep appearing in Ethereum smart contracts. Some of the vulnerabilities discussed in this paper are avoidable programming errors. Therefore, it is necessary to always be sensitive to vulnerabilities when writing smart contracts. Multiple audits and checks need to be performed to avoid vulnerabilities before deploying the contract to the blockchain.

In addition to the known vulnerabilities summarized in this paper, new vulnerabilities may continue to appear in Ethereum smart contracts. Therefore, developing vulnerability-free smart contracts remains a challenge. Future work will involve detecting new vulnerabilities and avoiding potential vulnerabilities from the coding design to mitigate the significant security vulnerabilities presented in this paper.

REFERENCES

[1] A. M. Antonopoulos and G. Wood, Mastering ethereum: building smart contracts and dapps, O'reilly Media, 2018.
[2] R. M. Parizi, A. Amritraj, and Dehghantanha, Smart Contract Programming Languages on Blockchains: An Empirical Evaluation of Usability and Security, Blockchain - ICBC 2018, Cham, S. Chen, H. Wang, and and L.-J. Zhang, Eds. Springer International Publishing, 2018.
[3] E. Banisadr. [Online]. Available: https://medium.com/@ebanisadr/how-800k-evaporated-from-the-powh-coin-ponzi-scheme-overnight-1b025c33b530(accessed
[4] [Online]. Available: https://etherscan.io/address/0x741f1923974464efd0aa70e77800ba5d9ed18902#code(accessed
[5] J. Sanjuas. [Online]. Available: https://medium.com/coinmonks/an-analysis-of-a-couple-ethereum-honeypot-contracts-5c07c95b0a8d
[6] C. Coverdale. [Online]. Available: https://medium.com/coinmonks/solidity-tx-origin-attacks-58211ad95514
[7] 2019.
[8] A. Molina. [Online]. Available: https://medium.com/coinmonks/smart-contracts-common-vulnerabilities-solidity-e64c5506b7f4
[9] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)," in Principles of Security and Trust, M. Maffei and M. Ryan, Eds. Springer, 2017, pp. 164–186.
[10] S. Palladino. [Online]. Available: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/(accessed
[11] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," ACM Computing Surveys (CSUR), vol. 53, no. 3, pp. 1–43, 2020.
[12] [Online]. Available: https://consensys.github.io/smart-contract-best-practices/attacks/denial-of-service/(accessed
[13] [Online]. Available: https://swcregistry.io/docs/SWC-113(accessed
[14] M. H. Swende. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1884(accessed
[15] A. Averin and O. Averina, "Review of blockchain technology vulnerabilities and blockchain-system attacks," 2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon), pp. 1–6, 2019.
[16] A. M. Antonopoulos, 2014.
[17] A. Studnev. [Online]. Available: https://bitquery.io/blog/ethereum-classic-attack-8-august-catch-me-if-you-can(accessed
[18] pp. 20–20. [Online]. Available: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/#:~:text=The%20threat%20of%20a%2051
[19] Cnbctv18 and Com. [Online]. Available: https://www.cnbctv18.com/technology/explained-replay-attacks-and-how-they-can-affect-blockchains-14328412.htm(accessed
[20] Tiana. [Online]. Available: https://coin98.net/what-is-replay-attack(accessed
[21] Q. Labs. [Online]. Available: https://quantstamp.com/blog/preventing-replay-attacks-post-ethereum-merge(accessed
[22] S. Eskandari, S. Moosavi, and J. Clark, "SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain," Financial Cryptography and Data Security, pp. 170–189, 2020.
[23] J. Zmysłowski. [Online]. Available: https://www.securing.pl/en/front-running-attack-in-defi-applications-how-to-deal-with-it/(accessed
[24] Z. Oualid. [Online]. Available: https://www.getsecureworld.com/blog/what-is-timestamp-dependence-vulnerability/(accessed
[25] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum project yellow paper, vol. 151, pp. 1–32, 2014.
[26] Onur. [Online]. Available: https://solmaz.io/2018/10/18/anatomy-block-stuffing/(accessed
[27] Y. Kulkarni. [Online]. Available: https://blog.finxter.com/randomness-or-replicatedlogic-attack-on-smart-contracts/(accessed
[28] Y. Riady. [Online]. Available: https://yos.io/2018/10/20/smart-contract-vulnerabilities-and-how-to-mitigate-them/#vulnerability-bad-randomness(accessed