Smart Contract Vulnerability Analysis and Security Audit

Daojing He, Zhi Deng, Yuxing Zhang, Sammy Chan, Yao Cheng, and Nadra Guizani

Abstract

Ethereum started the blockchain-based smart contract technology that due to its scalability more and more decentralized applications are now based on. On the downside this has led to the exposure of more and more security issues and challenges, which has gained widespread attention in terms of research in the field of Ethereum smart contract vulnerabilities in both academia and industry. This article presents a survey of the Ethereum smart contract's various vulnerabilities and the corresponding defense mechanisms that have been applied to combat them. In particular, we focus on the random number vulnerability in the Fomo3d-like game contracts, as well as that attack and defense methods applied. Finally, we summarize the existing Ethereum smart contract security audit methods and compare several mainstream audit tools from various perspectives.

Introduction

In 1997, Szabo [1] proposed the concept of smart contract. He believed that with the advancement of computer technology, smart contracts could be built through the use of algorithms and computer networks. Smart contracts do not need human intervention; instead, people only need to write rules according to the requirements and deploy them into the network. Algorithms can ensure that the contracts are executed correctly according to the contracting rules and guarantee their observability, verifiability, privity, and enforceability.

The Bitcoin [2] system is a digital currency system based on cryptography. It uses a public-private key pair to represent a user. The public key is equivalent to an ID number and is used to uniquely identify a Bitcoin user. The private key is kept by the user and it is the only credential recognized by the system that the user owns the public key. The Bitcoin system consists of all individual users and has no centralized management. Each individual in the Bitcoin system keeps the same ledger, which records all the transactions since the system was launched. The ledger is in blocks. Each block is a data block in which transaction information is recorded. In each block, the hash value of its previous block is also stored. In this way, a linked list of blocks is constructed, hence called a "blockchain". In addition, the system ensures that the ledgers recorded by each member are completely consistent by running a consensus mechanism called proof-of-work (PoW). By using the Bitcoin system, one can avoid the high cost and security risks brought by centralized management.

The Bitcoin system realizes decentralized secure transactions through a hash algorithm, the PoW consensus mechanism, and a blockchain data structure. The PoW consensus mechanism guarantees privity. Each user can query all transaction records since the system has been created, ensuring its observability and verifiability. Unless a user or a group of users controls more than 50 percent of the local computing power on mining, it is not possible to tamper the data on the blockchain, and the system will run normally according to the rules. As the system currently has a huge number of miners, it is very difficult for malicious attackers to take up more than 50 percent of mining power, which guarantees its enforceability.

The drawback of the Bitcoin system is that the functionality is relatively simple. The rule was set by a system creator in the beginning and can only be used for money transfer between users. Economically, it does not make sense for each application that uses Bitcoin to have a new Bitcoin system setup.

The Ethereum system proposed in 2015 perfectly realized the concept of smart contract for the first time. Ethereum is a cryptocurrency system launched by Vitalik Buterin [3, 4]. Ethereum combines smart contracts with blockchain so that any user can use Ethereum to create smart contracts, not just for the purpose of money transfer.

Ethereum divides accounts into externally owned accounts and contract accounts. An externally owned account has a pair of public and private keys that can be used for transfer operations. The contract account has only a contract address without a private key. Note that users cannot directly transfer the contract account balance due to the lack of a private key. The contract account can only be created by an externally owned account. Users can first use the Ethereum smart contract development language, *Solidity*, to develop contracts. Then users can deploy the compiled bytecode, that is, the contract, to the blockchain. Once deployed, the Ethereum system will automatically generate a unique address for the contract. This address is called a contract address. A user can call the functions in a contract using its contract address. In this way, all users can implement their own smart contracts based on Ethereum.

With the increasing popularity of Ethereum, a variety of smart contracts and smart-contract-based Dapps (Decentralized applications) have been developed. In 2015, the cat game Crypto-

Digital Object Identifier:
10.1109/MNET.001.1900656

Daojing. He, Zhi Deng, and Yuxing Zhang are with East China Normal University; Sammy Chan is with City University of Hong Kong; Yao Cheng is with the Institute for Infocomm Research; Nadra Guizani is with Purdue University.

0890-8044/20/$25.00 © 2020 IEEE

Kitties was warmly welcomed as the cats raised in the blockchain cannot be stolen. So far, the number of Dapps based on Ethereum smart contracts have reached 1,228. In addition, Ethereum also supports users to easily develop their own token contracts and issue their own digital tokens. The launch of the ERC-20 standard for implementing tokens on Ethereum blockchain prompted a surge in the number of tokens on Ethereum. Currently, the total amount of tokens raised through the ICO has reached $7,812,150,041 [5], and token types have reached 1,253.

Recently, security issues in Ethereum smart contracts have been exposed and attracted attention from both academia and industry. This article presents a survey of the vulnerabilities as well as the defenses that have been taken. In the following section, we analyze the common vulnerabilities in Ethereum smart contracts, and give some examples. Then we explain the random number vulnerability in Fomo3d-like games and give some recommended remedies. Finally, we summarize some famous smart contract audit tools.

Ethereum Smart Contract Vulnerabilities

Due to the distributed nature of blockchain, once the contract is deployed, the code cannot be modified. As a result, if there is any vulnerability in the deployed contract, especially the smart contracts that manage a lot of Ethereum tokens, it can lead to very serious consequences. In 2016 the "DAO" incident became the most well known security breach. By using the reentrancy vulnerability, hackers stole 3.6 million Ethereum tokens, worth 50 million US dollars. In 2017, hackers exploited the vulnerability in Parity Multi-Signature Wallet to steal 150,000 Ethereum tokens, worth 30 million US dollars. In August 2018, hackers used the vulnerability in random number generation in Fomo3d and LastWinner to obtain Ethereum tokens that were worth 200 million US dollars.

Security vulnerabilities in smart contracts can be divided into the following categories.

Unchecked Send/Other Function Call

When an important function such as *send* or *transfer* is called, there must be a proper check for function results before proceeding to the subsequent function logic. Unexpected program branching and loss of contract result may happen if there is no proper check of function results. For example, the RichMan contract and the Bank contract in Contract 1 form a system evaluating the wealth of an account holder, based on the amount of deposits in the bank. A user who wants to be classified as rich needs to transfer a sum of ETH (Ethereum digital currency abbreviation) through the bank to a current rich contract address, and the transfer amount must be greater than the current "treasure price". After a successful transfer, the rich man title is assigned to the transferor and the treasure price becomes the transfered amount. However, the RichMan contract fails to check whether the transfer is successful before assigning the rich man title. As a consequence, an attacker can make a failed transfer by bidding a price that is much larger than the balance in the bank, so that the attacker can obtain the rich man title without sending the treasure price.

```
contract RichMan {
    address richman;
    uint treasure ;
    Bank bank;
    constructor (uint value , address bank_addr)public{
        bank = Bank(bank_addr);
        if (bank.getDeposit (msg.sender) >= value){
            richman = msg.sender;
            treasure = value;
        }
    }
    function Apply(uint value) public {
        if (value > treasure ){
    // fail to check whether the transfer is
        successful
        bank.transferMoney(richman,value);
        treasure = value;
        richman = msg.sender;
        }
    }
}

contract Bank {
    mapping (address => uint256) balances;
     ......
    function withdraw(uint amount)public{
        if (balances[msg.sender] >= amount){
        msg.sender.call .value(amount)
        balances[msg.sender] -= amount;
        }
    }
    function transferMoney(address to,uint value)
    public returns (bool){
        if (balances[msg.sender]<value)
            return false ;
        else{
            balances[msg.sender] -= value;
            balances[to] += value;
            return true ;
        }
    }
    function SelfDestruct () onlyOwner{
        ......
        Destruct ()
    }
    function Destruct (){
        selfdestruct ();
    }
     ......
}
```

CONTRACT 1. RichMan contract.

Reentrancy Vulnerabilities

Reentrancy vulnerabilities are also caused by developer negligence [6]. We use the example in Contract 2 to explain how the reentrancy vulnerability works. In the Bank contract in Contract 1, the user is allowed to call the function *withdraw()* to extract all the ETHs stored in the bank. However, there is an anonymous fallback function in Ethereum, which would be called automatically once any mount of ETHs is transferred to that contract. The attacker can first deposit a quantity of ETHs into the bank through the attack function in the Attacker contract, and then extract it. When the bank transfers ETHs to the attacker's contract, the EVM (Ethereum virtual machine) will automatically execute the anonymous fallback function *payable()*, and now, the balance has not been subtracted. So in the fallback function, the program can successfully call the withdraw function again, and the EVM will execute the fallback function again. This creates an infinitie loop of reentrancy. To avoid "out of gas" exception, the attacker must limit the loop layers. In Contract 2, the number of layers is limited to 10. In this way, the attacker can continue to attack until all the Ethereum of the bank contract is extracted.

```
contract Attacker{
    .....
    uint256 times = 0;
    uint amount;
    function attack () public{
        Bank(msg.sender).deposit(amount);
        Bank(msg.sender).withdraw(amount);
    }
    function () payable{
        if (times < 10){
            times++;
            Bank(msg.sender).withdraw(amount);
        }
    }
}
```

CONTRACT 2. Attacker contract.

Permission Control Vulnerabilities

Since everything on the blockchain is public, once the function call permission is not effectively controlled, it may allow adversaries to call the high-privilege function, causing fatal damage to the contract. For example, if the function for changing the balance is not controlled, an attacker can arbitrarily manipulate the balance. More seriously, if it is the *selfDestruct()* function to which the access is not controlled, an attacker can destroy the contract and transfer the remaining tokens to the attacker's account.

This vulnerability arises mainly due to the following three reasons [7]:

Improper Use of msg.origin and msg.sender: Some smart contract developers do not understand the difference between *msg.origin* and *msg.sender*. *Msg.origin* returns the originator of the call, which must be a user address rather than a contract address. *Msg.sender* returns the direct caller of the function, which may be a user address or a contract address. When verifying the identity of the caller, if these functions are used incorrectly, the verification may lead to incorrect program branches.

Failure in Distinguishing Human from Contract: A contract is a program which can achieve more accurate operations than a human. For example, it is difficult for a human to forecast the value of *block.timestamp* precisely before the execution, because it would be executed on an uncertain miner. However we can call the *block.timestamp* in our own contract, and then call the target contract in our contract to predict the value of *block.timestamp*. Therefore, the failure in determining whether the smart contract caller is human or contract may lead to uncertain consequences. In the very popular Fomo3d game, the hacker can obtain huge profits through the airdrop function by accurately predicting the *block.timestamp* in a contract.

Spelling Mistake: Constructors are usually used for initialization and determining the owner of the contract. If a programmer misspells the constructor name during programming, such an error would not be detected by the compiler, causing the constructor to be a public function that everyone can call. For example, consider that the constructor of the contract *HelloWorld* is misspelled as *Helloworld*, any user can call the *Helloworld* function to change the owner of the contract.

Function Visibility Vulnerabilities

In Solidity, the default visibility property of a function is public. So when the developer forgets to declare visibility for a private function, the function is accessible to anyone. For example, in the Bank contract of Contract 1, anyone can call the *Destruct* function to directly destroy the contract.

Timestamp Dependency

Different from the traditional program, the execution environment of the smart contract is at the miner side. When some logic in the contract depends on the current time, the miner can control the current time to control the execution result to achieve a certain expectation.

Random Number Vulnerabilities

In a contract which uses a publicly known variable as a seed to generate a random number, an attacker can accurately predict the random number. We will discuss this vulnerability in detail in the next section using the airdrop vulnerability in the Fomo3d game contract as an illustrating example.

Integer Overflow/Underflow

Ethereum Solidity code is first compiled into machine code and executed in 256-bit EVM. Integer overflow is a potential risk. For a *uint256* type, the maximum value that can be represented is 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff. The value will be changed to 0 after adding 1 to the maximum value, and the the maximum value is obtained by subtracting 1 from 0. For example, in the *withdraw* function in the Bank contract of Contract 1, since it does not consider the possibility of an integer underflow, an attacker can make their account balance a huge number by extracting the ETHs over their balance.

Airdrop Vulnerability in Fomo3d

In this section, using Fomo3d's Airdrop function as an example, we analyze its security issues in detail.

Game Introduction

Fomo3d is a game with complex rules that runs on the Ethereum network [8]. The core element of the game is the *key*, which represents the number of assets the player has in the game. Each player participates in the game by buying a key. The price of a key will fluctuate slightly as the game progresses. There is a countdown timer in the game. In each round, players can purchase one or more keys at any time. Each time a player buys a key, the system will automatically increase the timer by 30 seconds. Part of the ETH that the player uses to purchase the key is used as dividends for the player who previously held the key, and part of it flows into the prize pool. When the timer returns to 0, the last player who purchased the key can get half of the prize in the prize pool, and the rest will be assigned to the other players holding the key and the starting capital for the next round according to certain rules. The game will automatically enter the next round.

In addition to the prize pool and dividends, Fomo3d also uses the airdrop mechanism to stimulate players to purchase keys. One percent of the amount of buying keys will be allocated into

the airdrop prize pool. At the beginning of each round of the game, the player's chance of getting an airdrop is 0, and each order of more than 0.1 ETH will increase the chance of getting the airdrop by 0.1 percent. Once a player gets an airdrop, the probability that all players will get an airdrop will return to zero. The amount of money a player receives for an airdrop depends on the amount of ETH spent on the purchase. For a purchase of 0.1–1 ETH, there is a 25 percent chance of winning the airdrop side pot. For a purchase of 1–10 ETH, there is a 50 percent chance of winning the airdrop side pot. For a purchase of more than 10 ETH, there is a 75 percent chance of winning the airdrop side pot. The vulnerability introduced in this game appears in the game's airdrop mechanism.

Vulnerability Analysis

Random Number Vulnerability: Similar to the traditional lottery mechanism, Fomo3d also determines whether to win by generating a random number. Contract 3 shows the function that generates the random number. It uses *block.timestamp*, *block.difficulty*, *now*, *block.coinbase*, *block.gaslimit*, *msg.sender*, and *block.number* as the seed for generating random numbers. *Block.timestamp* and *now* refer to the current time, *block.coinbase* refers to the miner's address, *block.difficulty* refers to the number of hashes that need to be calculated for the current block, and *block.gaslimit* refers to the maximum gas limit of the current block. *Msg.sender* refers to the caller of the function. *Block.number* refers to the current block number.

However, these variables are public on the blockchain, so the attacker can calculate the winning award by using the algorithm to see if the number of rewards exceeds the cost. If a profit can be made, then he makes a purchase. In this way, an attacker can continue to profit from the airdrop.

Identity Verification Vulnerability: In the contract of Fomo3d, in order to avoid an attacker calling the contract through a contract, a function modifier *isHuman()* is introduced in Contract 4 to restrict key purchasing ability to only humans. The modifier determines whether the caller is a contract by checking the caller's *codesize*. If *codesize* is not 0, it breaks and prompts that only humans can call the function. *Codesize* is an attribute of any address in Ethereum, referring to the number of bytes of executable code owned by the address. Normally, the *codesize* of a user's address is 0. For a contract, even if there is only one empty constructor, the *codesize* is never 0.

However, there is a huge loophole in the contract. The code will not be deployed until the contract constructor's execution finishes. In the contract constructor, the *codesize* is 0. So an attacker can make purchases in the contract constructor, bypassing the *isHuman()* modifier.

Attack Plan

Because of the above two vulnerabilities, [9] proposed the simplest attack scheme, as shown in Fig. 1a. First, the attacker writes the same code as the Fomo3d *airdrop()* in the constructor of the PwnFomo3d contract to generate a random number, and calculates whether the random number can win. If the result is positive, the keys are purchased immediately, otherwise the program reverts to the calculation loop. However, this type of attack is inefficient. Because the attacker needs to deploy the contract every time they try to attack, and the deployment of the contract consumes a lot of gas.

```
function airdrop() private view returns(bool){
    uint256 seed = uint256(keccak256(abi.
        encodePacked(
        (block.timestamp).add
        (block.difficulty).add
        ((uint256(keccak256(abi.encodePacked(block.
            coinbase)))/(now)).add
        (block.gaslimit).add
        ((uint256(keccak256(abi.encodePacked(msg.
            sender)))/(now)).add
        (block.number)
        ));

    if((seed - ((seed / 1000) * 1000)) <
        airDropTracker_)
        return(true);
    else
        return(false);
}
```

CONTRACT 3. Airdrop function in Fomo3d.

```
modifier isHuman() {
    address _addr = msg.sender;
    uint256 _codeLength;
    assembly {_codeLength := extcodesize(_addr)}
    require(_codeLength == 0, "sorry␣humans␣
        only");
    _;
}

modifier isHuman1(bytes msg, bytes sign,bytes32 r,
    bytes32 s, byte v1) {
    uint8 v = uint8(v1) + 27;
    addr = ecrecover(msg, v, r, s);
    require(addr == msg.sender);
    _;
}

modifier isHuman2() {
    require(tx.origin == msg.sender);
    _;
}
```

CONTRACT 4. isHuman modifier.

There is a more cost-effective attack [10]. As shown in Fig. 1b, the attacker first deploys the Parent contract. Each time the attacker calls the attack function in the Parent contract, it predicts whether the caller will get an airdrop if he buys at that time. The parent contract deploys the contract to buy only if the prediction is positive. The difference between Fig. 1a and Fig. 1b is that the attacker in Fig. 1b predicts whether to get an airdrop in a fixed contract, which further reduces the cost of the attack. The difficulty of this attack method is to predict the address of the newly deployed PwnFomo3d Contract, which is a variable in the airdrop generating algorithm. This method just reduces the attack cost but does not improve the probability of a successful attack, so the attacker still needs to attack multiple times.

However, the attacker can solve this challenge by pre-deploying a number of sub-contracts as shown in Fig. 1c [11]. According to Fomo3d's rules, we know that the probability of a player getting an airdrop increases by 0.1 percent each time, that is, the probability of a player getting an

airdrop, is at least 1/1000. Therefore, the hacker first deploys 1000 sub-contracts through the parent contract and records the addresses of these sub-contracts. In each attack, the attacker traverses to choose a specific address from 1000 pre-deployed sub-contracts' addresses. In this way the attacker can get the airdrop calling that contract to purchase a key. In theory, the attacker increases the successful probability of each attack to 100 percent.

Defense Against Random Number Vulnerabilities

Through the above analysis, the vulnerability lies with the generated random number being predictable which may further lead to the failure in contract logic by attacking using another contract. Therefore, the defense against such a vulnerability can be achieved from two perspectives [12].

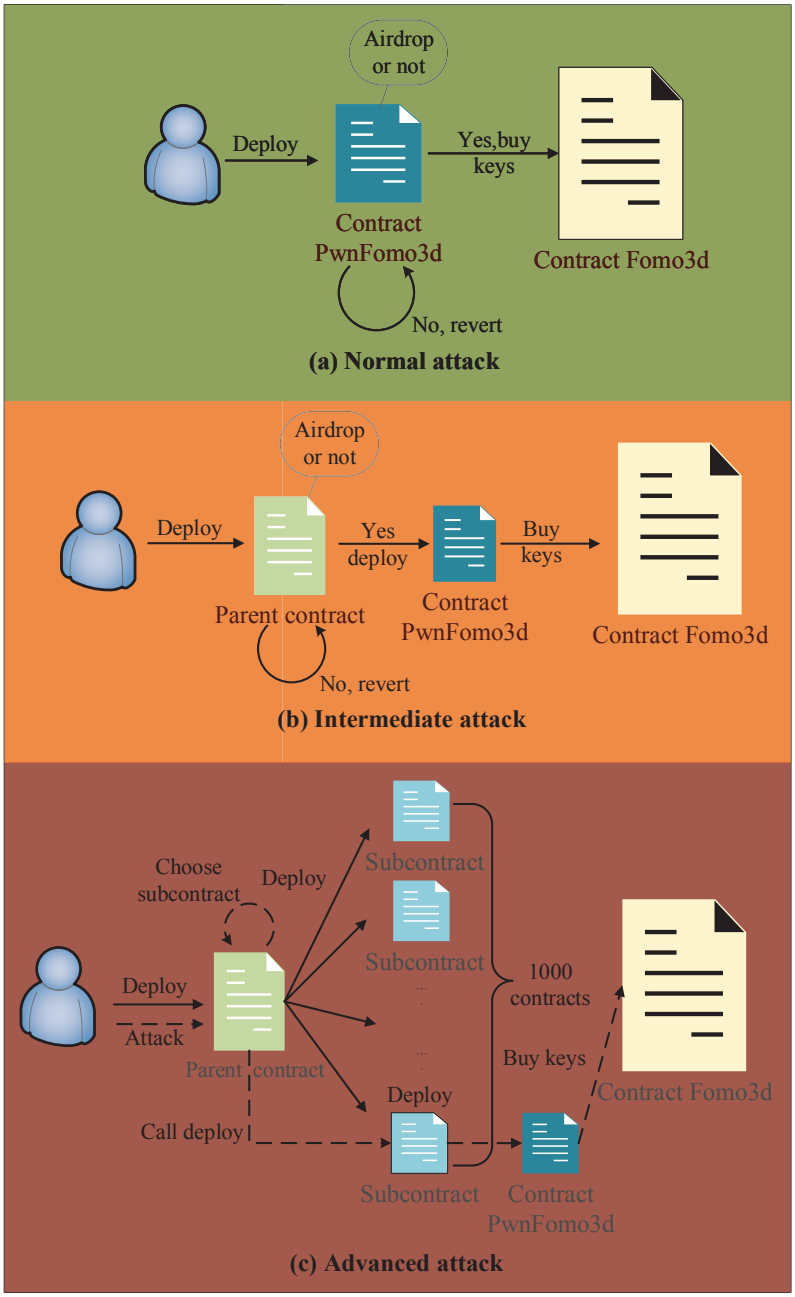

FIGURE 1. Fomo3d attack method.

Correctly Verify the Identity of the Caller:

Signature Verification: Using a signature is one solution in distinguishing between the human address and the contract address. In the Ethereum smart contract, each user has a private key corresponding to their own address. However, for the contract address, there is no private key. It is therefore possible to verify whether *msg.sender* has a private key by a challenge-response mechanism. The challenge is to request the sender signing a given message. If the signature verification is successful, it is considered that *msg.sender* owns the private key and is the user, otherwise the *msg.sender* is considered to be the contract. Ethereum provides a very convenient function *ecrecover()*, which returns the address corresponding to a signature. The developer can write a modifier, for example, *isHuman1* function in Contract 4, to verify the caller's signature, where msg refers to the SHA-3 value of the signed information, and *r*, *s*, *v1* refer to the data generated by the signature algorithm for the msg. However, the signing and verifying processes consume extra gas.

Origin == msg.sender: Another simpler and more cost effective solution is to determine whether *msg.sender* is equal to *tx.origin* as shown in the *isHuman2* function in Contract 4. When the attack is a via an attack contract to the victim contract, the *tx.origin* is the attacker while the *msg.sender* is the attack contract. The difference between these two values may indicate an attack exists.

Generating "True" Random Numbers: In addition to preventing contract calls, a fundamental solution is to generate pseudo random numbers that cannot be predicted. Instead of using public variables on the blockchain as seeds in random number generation, developers can use hash values of the future blocks or introduce external oracles such as Oraclize in the generation process. Ethereum allows the hash value of a specific block to be obtained by calling the *block.blockhash()* function. The random number seed is defined as the hash value of the current block, which is unpredictable before the current block is packed. The scheme divides the random number generation into two steps. First, the user needs to submit the request, and the contract records the block number of the user calling the contract. After the transaction is packaged, the hash value of the block number that was used in the first step is known. Using this hash value, one can make sure the random process, for example, the airdrop, is unpredictable.

Smart Contract Audit Method

Vulnerabilities in smart contracts can lead to great financial loss. Audit is a way to detect any security vulnerability before publishing the contract. Traditionally, auditing is done manually. There have been automated or semi-automated tools to do the contract audit. We introduce the manual audit process first, then we compare the advantages and disadvantages of existing tools.

Manual Audit

Manual auditing relies on the experience of security engineers. Due to the rapid development of smart contract technology, a qualified security engineer should keep up with the dynamics of

Tools	Features	Installation
Oyente (Python 2.7)	1. Analyze with symbolic execution 2. Analyze using EVM code or Solidity source code 3. Analyze using contract URL 4. Assertions checker for contract	1. Prepared docker 2. Pip install 3. Step by step install dependencies
Mythril (Python 3.5)	1. Analyze with concolic testing 2. Analyze using EVM code or Solidity code 3. Analyze using local contract or remote contract URL 4. CFG visualization 5. Blockchain exploration, to search specific contract, function calls, and EVM code sequeunce 6. Disassemble EVM bytescode or online code 7. Cross referencing of contract	1. Pip install 2. Download from github and install
Porosity (C++)	1. Analyze EVM code 2. Disassembe the EVM code 3. CFG visualization 4. Decompile the EVM code into Solidity	1. Compile by user 2. Windows release install package

TABLE 1. Comparison of smart contract audit tools.

smart contract security to keep abreast of the vulnerabilities exposed in EVM and new attacks. It requires the security engineers to be able to quickly read the code, combine the comments and even quickly clarify the contract logic without comments.

The advantage of manual auditing is the ability to understand the logic of the code from the code level and to identify logical loopholes. An excellent security engineer can keep up-to-date on various security intelligence and immediately check the contract to achieve a quick response. The drawback of manual auditing is that the auditing speed is limited and the process is expensive. With the increasing number of smart contract security issues exposed, smart contract code functions become more powerful, and architectures become more complex, it is increasingly difficult for human audits to more fully discover all the security issues in smart contracts. And because of the lack of reverse tools at this stage, security engineers cannot effectively audit closed-source contracts.

The Existing Audit Tools

With the development of a smart contract audit, some traditional vulnerability analysis methods such as symbolic execution, fuzzing test, and taint analysis are gradually being introduced into the audit of smart contracts. We introduce three tools, that is, Oyente, Mythril, and Porosity, and compare their advantages and disadvantages. The overall results are summarized in Table 1.

Oyente: Oyente [13] is an Ehtereum smart contract auditing tool based on symbolic execution, implemented by Python 2.7. It supports three different installation methods: docker installation, pip installation, manual compilation and installation. Oyente's architecture and detection methods for four different vulnerabilities are described in detail in [13]. It is able to detect TOD (Transaction-Ordering Dependence) vulnerabilities, timestamp vulnerabilities, and improper handling of exception vulnerabilities in smart contracts.

Oyente can run directly on the EVM bytecode. At the same time, the Solidity compiler is integrated. The input Solidity source code can be compiled into EVM bytecode and analyzed. Oyente also supports input contract URL for analysis and can check the assertion in the code. Oyente's input includes the contract code to be analyzed and the current Ethereum global state.

Mythril: Mythril [14] is a security analysis tool for analyzing EVM bytecodes. It is based on Python 3.5 and supports pip install and source code compilation and installation.

Mythril can detect security vulnerabilities built on platforms such as Ethereum, Quorum, Vechain, Roostock, Tron, and so on, using symbolic execution, SMT solution and taint analysis. The core module of Mythril is a smart contract virtual machine LARSRE. For the input smart contract, it outputs a graph, the graph node contains the currently executing code block, and the current program counter status. At the same time, the node also includes a group. The edge of the graph also contains a set of constraints that represent requirements to transition between different code blocks.

Mythril uses the concolic test method to detect smart contract's vulnerabilities. Similar to Oyente, it supports analysis of Solidity source code, EVM bytecode and contract URL.

In addition to the security analysis features described above, Mythril supports the following features:

- CFG (Control Flow Graph) visualization.
- Contract query function, which is able to query a specific function to call the latter EVM bytecode sequence.
- Disassemble local or online EVM code.
- Analyze the cross-reference relationship between different contracts.

However, Mythril can only audit the vulnerabilities of smart contracts according to existing rules. It cannot audit the correctness of smart contract functions. In practice, one of the sources of a large number of vulnerabilities is that the function of the smart contract does not match the actual design. At the same time, some vulnerabilities cannot extract valid constraint characteristics, which leads to false negatives.

Porosity: Porosity [15] is a reverse engineering tool that can decompile EVM code into Solidity source code, followed by some static analysis and dynamic analysis capabilities.

It can directly analyze EVM bytecode and supports the following functions:

- Analyze contract ABI (Application Binary Interface), listing all functions of the contract.
- Disassemble contract bytecode.
- CFG visualization.
- Decompile bytecode to Solidity source code.

Our team is committed to improving the existing smart contract auditing tools, combining dynamic auditing with static auditing, and proposing different dynamic and static auditing methods for various vulnerabilities.

As a decompilation tool, the vulnerability analysis capability of Porosity is not strong, but the decompilation function can provide the malicious attack events for security engineers, and the backtracking hacking methods bring great convenience.

Conclusion and Future Direction

This article has introduced the security issues in Ethereum's smart contracts, and focused on the random number vulnerability and verification vulnerability in the Fomo3d game. It has highlighted the importance of the smart contract security audit. This article also compared the pros and cons of commonly used audit tools. At the same time, our team is committed to improving the existing smart contract auditing tools, combining dynamic auditing with static auditing, and proposing different dynamic and static auditing methods for various vulnerabilities.

Acknowledgment

This research is supported by the National Key R&D Program of China (2017YFB0802805 and 2017YFB0801701); the National Natural Science Foundation of China (Grants: U1936120 and U1636216); the Joint Fund of the Ministry of Education of China for Equipment Preresearch (No. 6141A020333); the Shanghai Knowledge Service Platform for Trustworthy Internet of Things (No. ZF1213); the Fok Ying Tung Education Foundation of China (Grant 171058); and the Fundamental Research Funds for the Central Universities. Daojing He is the corresponding author of this article.

References

[1] N. Szabo, "Formalizing and Securing Relationships on Public Networks," *First Monday*, vol. 2, no. 9, 1997.
[2] S. Nakamoto *et al.*, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.
[3] W. Wang *et al.*, "Contract-Ward: Automated Vulnerability Detection Models for Ethereum Smart Contracts," *IEEE Trans. Network Science and Engineering*, DOI: 10.1109/TNSE.2020.2968505, Jan. 2020.
[4] G. Wood *et al.*, "Ethereum: A Secure Decentralised Generalised Transaction Ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, 2014, pp. 1–32.
[5] "ICOdata — ICO 2018 Statistics," https://www.icodata.io/stats/2018; accessed on 11/24/2019.
[6] C. Liu *et al.*, "Reguard: Finding Reentrancy Bugs in Smart Contracts," *Proc. 40th Int'l. Conf. Software Engineering: Companion Proc.*, ACM, 2018, pp. 65–68.
[7] L. Brent *et al.*, "Vandal: A Scalable Security Analysis Framework for Smart Contracts," arXiv preprint arXiv:1809.03981, 2018.
[8] S. Eskandari, S. Moosavi, and J. Clark, "Sok: Transparent Dishonesty: Front-Running Attacks on Blockchain," arXiv preprint arXiv:1902.05164, 2019.
[9] "How to pwn fomo3d, A Beginners Guide : Ethereum," https://www.reddit.com/r/ethereum/comments/916xni/how_to_pwn_fomo3d_a_beginners_guide/; accessed on 11/24/2019.
[10] "Fomo 3d Exploit Improved Clearly Explained : ethdev," https://www.reddit.com/r/ethdev/comments/91fpqd/fomo_3d_exploit_improved_clearly_explained/; accessed on 11/24/2019.
[11] "Largest smart contract attacks in blockchain history exposed ł part 1," https://medium.com/@AnChain.AI/largest-smart-contract-attacks-in-blockchain-history-exposed-part-1-93b975a374d0; accessed on 11/24/2019.
[12] "A Comprehensive Solution to Bugs in fomo3d-Like Games — by," https://hackernoon.com/a-comprehensive-solution-to-bugs-in-fomo3d-like-games-ab3b054f3cc5; accessed on 11/24/2019.
[13] L. Luu *et al.*, "Making Smart Contracts Smarter," Proc. 2016 ACM SIGSAC Conf. Computer Commun. security, ACM, 2016, pp. 254–69.
[14] "Consensys/mythril: Security Analysis Tool for Evm Bytecode, Supports Smart Contracts Built for ethereum, quorum, vechain, roostock, tron and other evm-Compatible Blockchains," https://github.com/ConsenSys/mythril; accessed on 11/24/2019.
[15] M. Suiche, "Porosity: A Decompiler for Blockchain-Based Smart Contracts Bytecode," DEF con, vol. 25, 2017, p. 11.

Biographies

Daojing He [S'07, M'13] received the B.Eng. (2007) and M. Eng. (2009) degrees from Harbin Institute of Technology, China, and the Ph.D. degree (2012) from Zhejiang University, China, all in computer science. He is currently a professor in the School of Software Engineering, East China Normal University, P.R. China. His research interests include network and systems security. He is on the editorial board of several international journals such as *IEEE Communications Magazine*.

Zhi Deng is a master student in the School of Software Engineering, East China Normal University, P.R. China.

Yuxing Zhang is a Ph.D. student in the School of Software Engineering, East China Normal University, P.R. China.

Sammy Chan [S'87, M'89] received his B.E. and M.Eng. Sc. degrees in electrical engineering from the University of Melbourne, Australia, in 1988 and 1990, respectively, and a Ph.D. degree in communication engineering from the Royal Melbourne Institute of Technology, Australia, in 1995. Since December 1994 he has been with the Department of Electrical Engineering, City University of Hong Kong, where he is currently an associate professor.

Yao Cheng received her Ph.D. degree from University of Chinese Academy of Sciences in 2015. She was a research fellow at the School of Information Systems, Singapore Management University until 2017. She is currently a scientist at the Institute for Infocomm Research, A*STAR, Singapore. Her research interests are in the information security area, focusing on vulnerability analysis, privacy leakage and protection, malicious application detection, usable security solutions, and AI applications in security areas.

Nadra Guizani is a clinical assistant professor and a Ph.D. candidate at the School of Electrical and Computer Engineering, Purdue University. Her research interests include machine learning, mobile networking, large data analysis, and prediction techniques. She is an active member of both the Women in Engineering program and the Computing Research Association.