Contract Audit
SymX

Smart Contract
Security Audit Report

Num: 07311750181151

Date: 2024-07-31

Welcome to SymX!

0x01 Summary Information

The SymX platform received this smart contract security audit application and audited the contract in Jul 2024.

It is necessary to declare that SymX only issues this report in respect of facts that have occurred or existed before the issuance of this report, and undertakes corresponding responsibilities for this. For the facts that occur or exist in the future, SymX is unable to judge the security status of its smart contract, and will not be responsible for it. The security audit analysis and other content made in this report are based on the documents and information provided to smart analysis team by the information provider as of the issuance of this report (referred to as "provided information"). SymX hypothesis: There is no missing, tampered, deleted or concealed information in the mentioned information. If the information that has been mentioned is missing, tampered with, deleted, concealed or reflected does not match the actual situation, SymX shall not be liable for any losses and adverse effects caused thereby.

Table 1 Contract audit information

Project	Description
Contract name	RealOldFuckMaker
Contract type	Ethereum contract
Code language	Solidity
Contract files	07311750181151.sol
Contract address	
Auditors	SymX team
Audit time	2024-07-31 17:50:20
Audit tool	SymX

Table 1 shows the relevant information of this contract audit in detail. The details and results of the contract security audit will be introduced in detail below.

0x02 Contract Audit Results

2.1 Vulnerability Distribution

The severity of vulnerabilities in this security audit is distributed according to the level of impact and confidence:

Table 2 Overview of contract audit vulnerability distribution

Vulnerability security distribution				
High	Medium	Low	Info	Opt
0	1	0	1	0

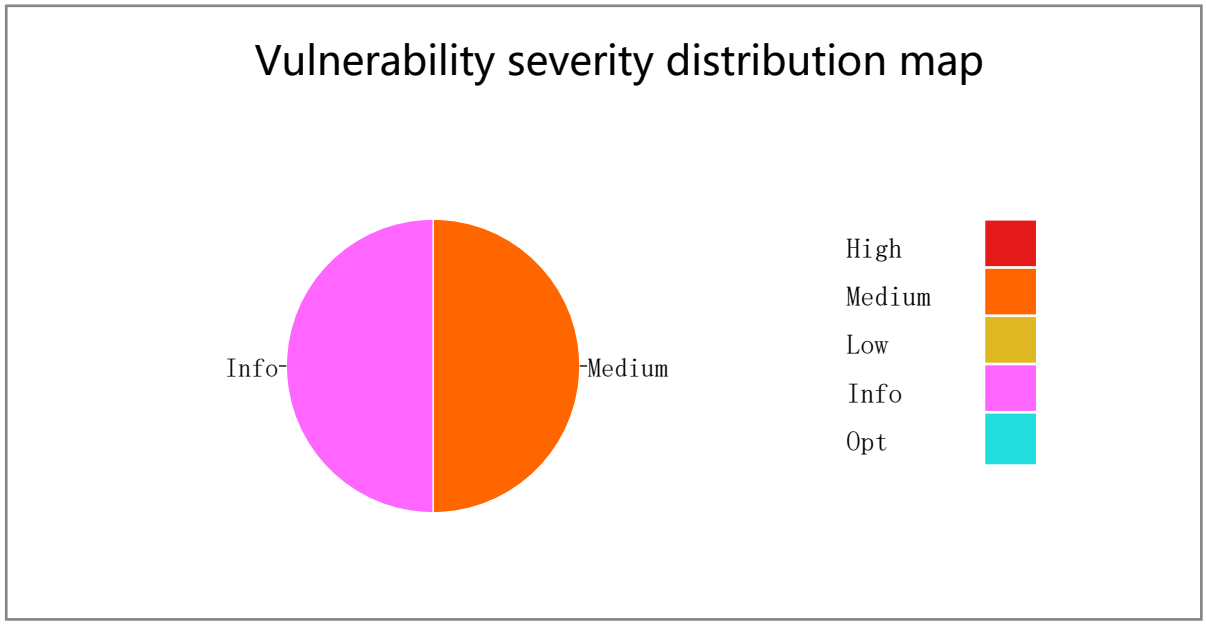

Figure 1 Vulnerability security distribution map

This security audit found 0 High-severity vulnerabilities, 1 Medium-severity vulnerabilities, 0 Low-severity vulnerabilities, 0 Optimization-severity vulnerabilities, and 1 places that need attention.

2.2 Audit Results

There are 18 test items in this security audit, and the test items are as follows (other unknown security vulnerabilities are not included in the scope of responsibility of this audit):

Table 3 Contract audit items

ID	Pattern	Description	Severity	Confidence	Status/Num
1	reentrancy-eth	Re-entry vulnerabilities (Ethereum theft)	High	probably	Pass
2	suicidal	Check if anyone can break the contract	High	exactly	Pass
3	controlled-delegatecall	The delegate address out of control	High	probably	Pass

4	arbitrary-send	Check if Ether can be sent to any address	High	probably	Pass
5	uninitialized-state	Check for uninitialized state variables	High	exactly	Pass
6	uninitialized-storage	Check for uninitialized storage variables	High	exactly	Pass
7	tod	Transaction sequence dependence for receivers/ethers	High	probably	Pass
8	incorrect-equality	Check the strict equality of danger	Medium	exactly	Pass
9	integer-overflow	Check for integer overflow	Medium	probably	Pass
10	unchecked-lowlevel	Check for uncensored low-level calls	Medium	probably	Medium:1
11	unchecked-send	Check unreviewed send	Medium	probably	Pass
12	tx-origin	Check the dangerous use of tx.origin	Medium	probably	Pass
13	timestamp	The dangerous use of block.timestamp	Low	probably	Pass
14	block-other-parameters	Hazardous use variables (block.number etc.)	Low	probably	Pass
15	low-level-calls	Check low-level calls	Info	exactly	Info:1
16	msgvalue-equals-zero	The judgment of msg.value and zero	Info	exactly	Pass
17	send-transfer	Check Transfe to replace Send	Opt	exactly	Pass
18	boolean-equal	Check comparison with boolean constant	Opt	exactly	Pass

0x03 Contract Code

3.1 Code

```
pragma solidity 0.4.24;

contract RealOldFuckMaker {
    address fuck = 0xc63e7b1DEcE63A77eD7E4Aeef5efb3b05C81438D;

    function makeOldFucks(uint32 number) {
        uint32 i;
        for (i = 0; i < number; i++) {
            //   UNCHECKED_LL_CALLS
            fuck.call(bytes4(sha3("giveBlockReward()")));
        }
    }
}
```

3.2 Contract CFG

The 1-th contract RealOldFuckMaker

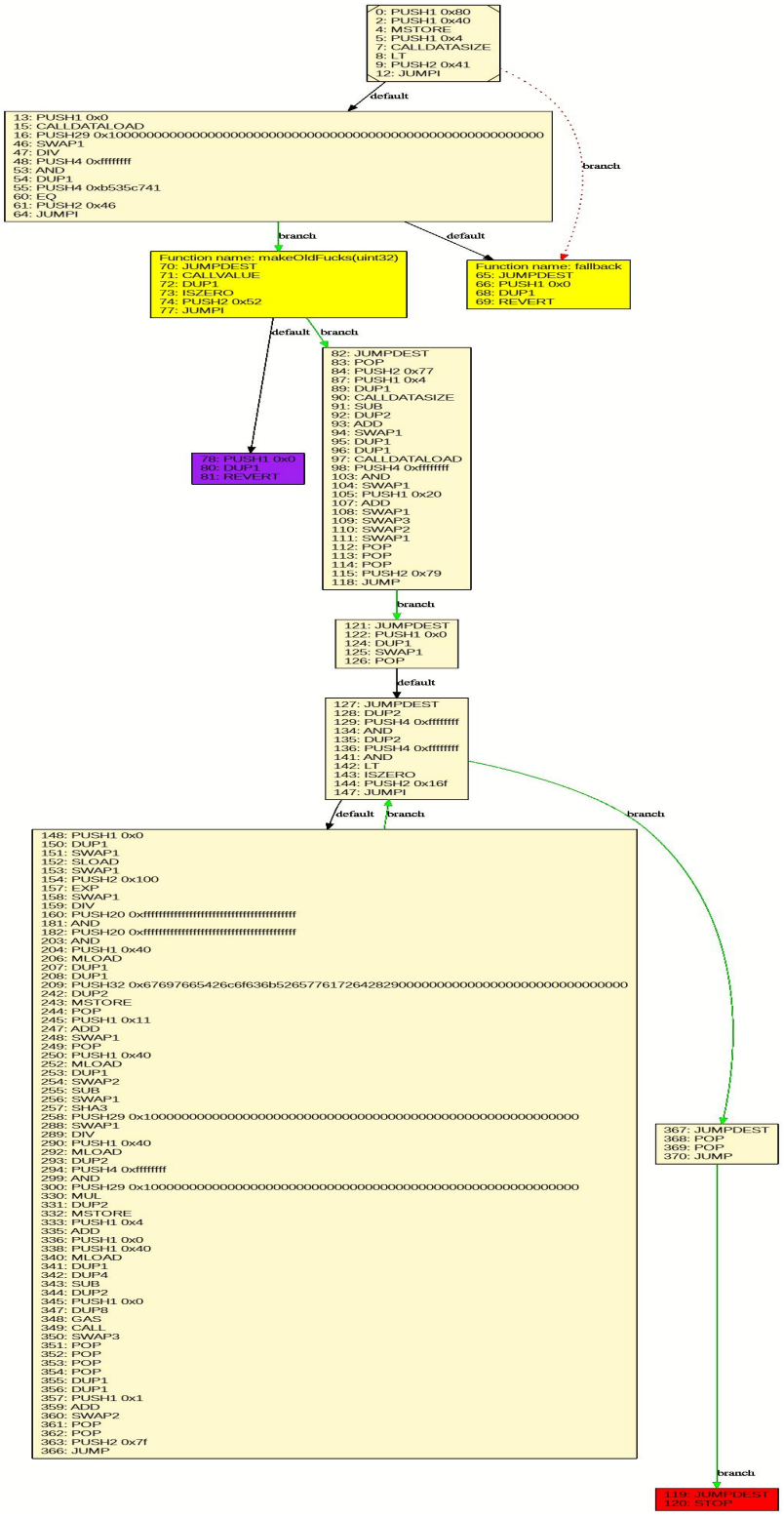

Figure 2 The CFG of contract RealOldFuckMaker.

0x04 Contract Audit Details

4.1 reentrancy-eth

Vulnerability description

A reentrancy error was detected. This is the reentry of ether. Through re-entry, the account balance can be maliciously withdrawn, resulting in losses. Do not report re-reporting that does not involve Ether (please refer to "reentrancy-no-eth")

Applicable scenarios

```
function withdrawBalance(){
   // send userBalance[msg.sender] Ether to msg.sender
   // if mgs.sender is a contract, it will call its fallback function
   if( ! (msg.sender.call.value(userBalance[msg.sender])() ) ){
      throw;
   }
   userBalance[msg.sender] = 0;
}
```

Bob used the reentrance vulnerability to call `withdrawBalance` multiple times and withdrew more than he originally deposited into the contract.

Audit results:【Pass】

Security advice: none.

4.2 suicidal

Vulnerability description

Due to lack of access control or insufficient access control, malicious parties can self-destruct the contract. Calling selfdestruct/suicide lacks protection.

Applicable scenarios

```
contract Suicidal{
   function kill() public{
      selfdestruct(msg.sender);
   }
}
```

Bob calls the "kill" function and breaks the contract.

Audit results:【Pass】

Security advice: none.

4.3 controlled-delegatecall

Vulnerability description

Delegate the call or call code to an address controlled by the user. The address of Delegatecall is not necessarily trusted, it is still a problem of access control, and the address is not checked.

Applicable scenarios

```
contract Delegatecall{
    function delegate(address to, bytes data){
        to.delegatecall(data);
    }
}
```

Bob calls `delegate` and delegates the execution of the malicious contract to him. As a result, Bob withdraws the funds from the contract and destroys the contract.

Audit results:【Pass】

Security advice: none.

4.4 arbitrary-send

Vulnerability description

The call to the function that sends Ether to an arbitrary address has not been reviewed.

Applicable scenarios

```
contract ArbitrarySend{
    address destination;
    function setDestination(){
        destination = msg.sender;
    }
    function withdraw() public{
        destination.transfer(this.balance);
    }
}
```

Bob calls setDestination and withdraw, and as a result, he withdraws the balance of the contract.

Audit results:【Pass】

Security advice: none.

4.5 uninitialized-state

Vulnerability description

Uninitialized state variables can lead to intentional or unintentional vulnerabilities.

Applicable scenarios

```
contract Uninitialized{
    address destination;
    function transfer() payable public{
        destination.transfer(msg.value);
    }
}
```

Bob calls "transfer". As a result, the ether is sent to the address "0x0" and is lost.

Audit results:【Pass】

Security advice: none.

4.6 uninitialized-storage

Vulnerability description

Local variables are not initialized. The initialized storage variable will be used as a reference to the first state variable, and key variables can be overwritten.

Applicable scenarios

```
contract Uninitialized{
   address owner = msg.sender;
   struct St{
      uint a;
   }
   function func() {
      St st;
      st.a = 0x0;
   }
}
```

Bob calls `func`. As a result, "owner" is overwritten with "0".

Audit results:【Pass】

Security advice: none.

4.7 tod

Vulnerability description

Mainly about the receiver's exception. A person who is running an Ethereum node can tell which transactions are going to occur before they are finalized. A race condition vulnerability occurs when code depends on the order of the transactions submitted to it.

Applicable scenarios

```
pragma solidity ^0.4.5;
contract StandardToken is ERC20, BasicToken {
   ...
   function approve(address _spender, uint256 _value) public returns (bool) {
      allowed[msg.sender][_spender] = _value;
      Approval(msg.sender, _spender, _value);
      return true;
   }
   ...
}
```

The execution result may be influenced by transaction orders.

Audit results:【Pass】

Security advice: none.

4.8 incorrect-equality

Vulnerability description

Using strict equality (== and !=), an attacker can easily manipulate these equality. Specifically: the opponent can forcefully send Ether to any address through selfdestruct() or through mining, thereby invalidating the strict judgment.

Applicable scenarios

```
contract Crowdsale{
   function fund_reached() public returns(bool){
      return this.balance == 100 ether;
   }
}
```

Crowdsale relies on fund_reached to know when to stop the sale of tokens. Bob sends 0.1 ether. As a result, fund_reached is always false, and crowdsale is always true.

Audit results:【Pass】

Security advice: none.

4.9 integer-overflow

Vulnerability description

When an arithmetic operation reaches the maximum or minimum size of the type, overflow/underflow will occur. For example, if a number is stored in the uint8 type, it means that the number is stored as an 8-bit unsigned number, ranging from 0 to 2^8-1. In computer programming, when an arithmetic operation attempts to create a value, an integer overflow occurs, and the value can be represented by a given number of bits-greater than the maximum value or less than the minimum value.

Applicable scenarios

```
contract Intergeroverflow{
   function bad() {
      uint a;
      uint b;
      uint c = a + b;
   }
}
```

The arithmetic result may be abnormal.

Audit results:【Pass】

Security advice: none.

4.10 unchecked-lowlevel

Vulnerability description

The low-level call to the external contract failed, and the return value was not judged. When sending ether at the same time, please check the return value and handle the error.

Applicable scenarios: 【Medium:1】

For this pattern, the specific problems in the contract are as follows:

The 1-th problem is located at pc 0x15e

The defective code locates at line [10], and it is shown as follows:

10 fuck.call(bytes4(sha3("giveBlockReward()")))

Security advice

Make sure to check or record the return value of low-level calls.

4.11 unchecked-send

Vulnerability description

Similar to unchecked-lowlevel, it is explained here that the return value of send and Highlevelcall is not checked.

Applicable scenarios

```
contract MyConc{
  function my_func(address payable dst) public payable{
    dst.send(msg.value);
  }
}
```

The return value of send is not checked, so if the send fails, the ether will be locked in the contract. If you use send to prevent block operations, please consider logging failed send.

Audit results:【Pass】

Security advice: none.

4.12 tx-origin

Vulnerability description

If a legitimate user interacts with a malicious contract, the protection based on tx.origin will be abused by the malicious contract.

Applicable scenarios

```
contract TxOrigin {
  address owner = msg.sender;
  function bug() {
    require(tx.origin == owner);
  }
}
```

Bob is the owner of TxOrigin. Bob calls Eve's contract. Eve's contract is called TxOrigin and bypasses the protection of tx.origin.

Audit results:【Pass】

Security advice: none.

4.13 timestamp

Vulnerability description

There is a strict comparison with block.timestamp or now in the contract, and miners can benefit from block.timestamp.

Applicable scenarios

```
contract Timestamp{
    event Time(uint);
    modifier onlyOwner {
        require(block.timestamp == 0);
        _;
    }
    function bad0() external{
        require(block.timestamp == 0);
    }
}
```

Bob's contract relies on the randomness of block.timestamp. Eve is a miner who manipulates block.timestamp to take advantage of Bob's contract.

Audit results:【Pass】

Security advice: none.

4.14 block-other-parameters

Vulnerability description

Contracts usually require access to time values to perform certain types of functions. block.number can let you know the current time or time increment, but in most cases it is not safe to use them. block.number The block time of Ethereum is usually about 14 seconds, so the time increment between blocks can be predicted. However, the lockout time is not fixed and may change due to various reasons (for example, fork reorganization and difficulty coefficient). Since the block time is variable, block.number should not rely on accurate time calculations. The ability to generate random numbers is very useful in various applications. An obvious example is a gambling DApp, where a pseudo-random number generator is used to select the winner. However, creating a sufficiently powerful source of randomness in Ethereum is very challenging. Using blockhash, block.difficulty and other

areas is also unsafe because they are controlled by miners. If the stakes arehigh, the miner can mine a large number of blocks by renting hardware in ashort period of time, select the block that needs to obtain the block hashvalue to win, and then discard all other blocks.

Applicable scenarios

```
contract Otherparameters{
   event Number(uint);
   event Coinbase(address);
   event Difficulty(uint);
   event Gaslimit(uint);
   function bad0() external{
      require(block.number == 20);
      require(block.coinbase == msg.sender);
      require(block.difficulty == 20);
      require(block.gaslimit == 20);
   }
}
```

The randomness of Bob's contract depends on block.number and so on. Eve is a miner who manipulates block.number and so on to use Bob's contract.

Audit results:【Pass】

Security advice: none.

4.15 low-level-calls

Vulnerability description

Label low-level methods such as call, delegatecall, and callcode, because these methods are easily exploited by attackers.

Applicable scenarios: 【Info:1】

For this pattern, the specific problems in the contract are as follows:

The 1-th problem is located at pc 0x15d

The defective code locates at line [10], and it is shown as follows:

10 fuck.call(bytes4(sha3("giveBlockReward()")))

Security advice

Avoid low-level calls. Check whether the call is successful. If the call is to sign a contract, please check whether the code exists.

4.16 msgvalue-equals-zero

Vulnerability description

msg.value==0 The check condition is meaningless in most cases.

Applicable scenarios

```
contract A{
    address owner;
    mapping(address => uint256) balances;
    constructor() {
        owner = msg.sender;
    }
    function B() return (uint256){
        if(masg.value == 0) {
            return 0;
        }
        balances[msg.sender] += msg.value;
        return balances[msg.sender];
    }
}
```

The condition "masg.value == 0" has no effect at most of the time.

Audit results:【Pass】

Security advice: none.

4.17 send-transfer

Vulnerability description

The recommended way to perform the check of Ether payment is addr.transfer(x). If the transfer fails, an exception is automatically raised.

Applicable scenarios

```
if(!addr.send(42 ether)) {
    revert();
}
```

The "send" operation can be replaced with "transfer".

Audit results:【Pass】

Security advice: none.

4.18 boolean-equal

Vulnerability description

Check the comparison of Boolean constants. There is no need to compare with true and false, so it's superfluous (gas consumption).

Applicable scenarios

```
contract A {
    function f(bool x) public {
        // ...
        if (x == true) { // bad!
         // ...
        }
        // ...
    }
}
```

```
Boolean constants can be used directly without comparison with true or false.
```

Audit results:【Pass】

Security advice: none.

Thanks for using the SymX contract audit platform!