LUNARAY
BLOCKCHAINSECURITY

SMART CONTRACT SECURITY AUDIT REPORT

for CyberCat

24 December 2021

lunaray.co

Table of Contents

1. Overview

On Dec 13, 2021, the security team of Lunaray Technology received the security audit request of the **CyberCat project**. The team completed the audit of the **CyberCat smart contract** on Dec 24, 2021. During the audit process, the security audit experts of Lunaray Technology and the CyberCat project interface Personnel communicate and maintain symmetry of information, conduct security audits under controllable operational risks, and avoid risks to project generation and operations during the testing process.

Through communicat and feedback with CyberCat project party, it is confirmed that the loopholes and risks found in the audit process have been repaired or within the acceptable range. The result of this CyberCat smart contract security audit: **Passed**

Audit Report Hash:

51E829CE419C5618522960A75FCEF601B6A6164E6DCBBD9AD9941E361DAE69AB

2. Background

2.1 Project Description

Project name	CyberCat
Contract type	Token , NFT , GameFi
Total Issue	2100 000 000 (HEP)
Code language	Solidity
Public chain	Binance , OKLink
Project address	https://www.cybercat.world/
Contract file	CoreCat.sol, CyberCatBlindBox.sol, HealthPotion.sol, RewardClaim.sol
Project Description	CyberCat is a Play-to-Earn mode crypto game published by CherrySwap. In the CyberCat world, players are able to collect, breed, raise, fight and trade the creatures known as CyberCats, and anyone can earn in CyberCat through skilled play and contributions to the ecosystem.

2.2 Audit Range

The smart contract file provided by CyberCat and the corresponding MD5：

Name	address
CoreCat.sol	D97874FA7222065AB727FEF65707AB2E
CyberCatBlindBox.sol	5206BA1101A54B4DDBDFA0393E111681
HealthPotion.sol	CC784CF730285796055A6D08C24C6717
RewardClaim.sol	03D7A62A710AD91C27E6D85B788A3334

3. Project contract details

3.1 Contract Overview

HealthPotion Contract

Issue HEP Token, $HEP (Health potion) is the CyberCat ecological currency, and HEP is an incentive token for players to participate in game competition.

CatCore Contract

The CatCore contract is the core logic of CyberCat. The main functions are to raise cats, form into adults, hatch, evolve cats, etc. In addition to this there are some variables that can be set by the administrator.

CyberCatBlindBox Contract

The CyberCatBlindBox contract is mainly for buying NFT, and three types of purchase exist. The other methods are mainly managed by onlyOwner, such as the setting of some variables.

RewardClaim Contract

The RewardClaim contract logic is mainly to send rewards, either by receiving the feeAmount yourself or by sending the RewardToken by the administrator.

3.2 Contract details

RewardClaim Contract

Name	Parameter	Attributes
RewardClaim	addresspayable _feeTo uint256 _feeAmount IERC20 _rewardToken	public
setFeeTo	addresspayable _feeTo	onlyOwner
setFeeAmount	uint256 _feeAmount	onlyOwner
setAdmin	address _admin bool _status	onlyOwner
claim	none	external
reward	address _user uint256 _amount uint256 _deadline	onlyAdmin

HasMinters Contract

Name	Parameter	Attributes
addMinters	address[] memory _addedMinters	onlyAdmin
removeMinters	address[] memory _removedMinters	onlyAdmin
isMinter	address _addr	public

ERC20Mintable Contract

Name	Parameter	Attributes
mint	address _to uint256 _value	onlyMinter
_mint	address _to uint256 _value	internal

CatCore Contract

Name	Parameter	Attributes
initialize	string _name string _symbol address _potion address _feeToken uint256 _feeAmount address _feeTo uint256 _feeToPercent address _feeToDead address _potionFeeTo address _geneScience uint256 _initSupply uint256 _matureDays	public
mintInit	uint256 _size	external
getCat	uint256 _catId	external
needPotions	uint8 _sireBreed uint8 _matronBreed	public
isBreedable	uint256 _sireId uint256 _matronId	public
setFeeInfo	address _potion address _feeToken uint256 _feeAmount address _feeTo address _potionFeeTo	external
setFeeToken	address _feeToken	external
setFeeAmount	uint256 _feeAmount	external
setFeeTo	address _feeTo	external
setFeeToDead	address _feeToDead	external
setFeeToPercent	uint256 _feeToPercent	external
setPotionFeeTo	address _potionFeeTo	external
setPotionFeeInfo	uint _index uint256 _amount	external
setBreedInfo	uint256 _breed uint256 _matureDays	external
setBreed	uint256 _breed	external

forbidToken	uint256 tokenId	external
unForbidToken	uint256 tokenId	external
setMatureDays	uint256 _matureDays	external
setNewGeneScience	address _geneScience	external
breedCat	uint256 _sireId uint256 _matronId	external
morphToAdult	uint256 _catId	external
_spawnCat	uint256 _gene address _owner	private
evolveCat	uint256 _catId uint256 _newGene	public
setBaseUri	string _uri	public
setUriPre	string _uriPre	public
setUriTail	string _uriTail	public
setTokenURI	uint256 _tokenId	internal

CyberCatBlindBox Contract

Name	Parameter	Attributes
initialize	IERC20 _cherryToken uint256 _chePrice IERC20 _usdtToken uint256 _usdtPrice uint256 _basePrice addresspayable _feeAddress uint256 _maxBoxAmount uint256 _maxSizeOneTimes address _nftAddress string _sec	onlyOwner
setMaxBoxAmount	uint256 _maxBoxAmount	onlyOwner
setMaxSizeOneTimes	uint256 _maxSizeOneTimes	onlyOwner
setSec	string _sec	onlyOwner
setChePrice	uint256 _chePrice	onlyOwner
setUsdtPrice	uint256 _usdtPrice	onlyOwner
setEthPrice	uint256 _ethPrice	onlyOwner
setFeeAddress	addresspayable _feeAddress	onlyOwner
_generateNextNFTId	none	private
_generateNextBOXId	none	private
_generateNextCollectionId	none	private
depositNFT	uint256 _tokenId	onlyOwner
withdrawNFT	uint256 _nftId	onlyOwner
withdraw	uint256 _collectionId uint256 _size	onlyOwner
buyByChe	uint256 _collectionId uint256 _size	external
buyByU	uint256 _collectionId uint256 _size	external

buyByEth	uint256 _collectionId uint256 _size	external
getBuyNum	uint256 _collectionId address _user	external
myBox	uint256 _collectionId	external
getCollectionNFTId	uint256 _collectionId	onlyOwner
getCurrentCollectionId	none	external
createCollection	string _name uint256 _size uint256 _durationBlock uint256 _cheLimitSize uint256 _uLimitSize uint256 _ethLimitSize	onlyOwner
getUnSaleNFT	uint256 _collectionId uint256 _limit	internal
setSaleLimit	uint256 _collectionId uint256 _cheLimitSize uint256 _uLimitSize uint256 _ethLimitSize	onlyOwner
setCheLimit	uint256 _collectionId uint256 _cheLimitSize	onlyOwner
setULimit	uint256 _collectionId uint256 _uLimitSize	onlyOwner
setEthLimit	uint256 _collectionId uint256 _ethLimitSize	onlyOwner
publishCollection	uint256 _collectionId uint256 _startBlock uint256 _durationBlock	onlyOwner
getBoxs	uint256 _collectionId	onlyOwner
isOnSale	uint256 _collectionId	public
randomIndex	uint256 _collectionId uint256 nonce uint256 _limitSize uint256 _type	internal
randomBoxIndex	uint256 _collectionId	internal

4. Audit details

4.1 Findings Summary

Severity	Found	Resolved	Acknowledged
● High	0	0	0
● Medium	0	0	0
● Low	2	0	2
● Info	1	0	1

4.2 Risk distribution

Name	Risk level	Repair status
Administrator Permissions	Low	Acknowledged
No events added	Info	Acknowledged
Claim Logic Analysis	Low	Acknowledged
breedCat Logic Analysis	No	normal
morphToAdult Logic Analysis	No	normal
_spawnCat Logic Analysis	No	normal
buyByU Logic Analysis	No	normal
Variables are updated	No	normal
Floating Point and Numeric Precision	No	normal
Default visibility	No	normal
tx.origin authentication	No	normal
Faulty constructor	No	normal
Unverified return value	No	normal
Insecure random numbers	No	normal
Timestamp Dependent	No	normal
Transaction order dependency	No	normal
Delegatecall	No	normal
Call	No	normal
Denial of Service	No	normal
Logical Design Flaw	No	normal
Fake recharge vulnerability	No	normal

Short address attack Vulnerability	No	normal
Uninitialized storage pointer	No	normal
Frozen account bypass	No	normal
Uninitialized	No	normal
Reentry attack	No	normal
Integer Overflow	No	normal

4.3 Risk audit details

4.3.1. Administrator Permissions

- **Risk description**

RewardClaim contract, CatCore contract, CyberCatBlindBox contract, HealthPotion contract, there are several methods of administrator privileges, can carry out sensitive operations, the current project has added Timelock contract, if the administrator private key is controlled by malicious people, or can lead to abnormal loss of funds and shake the stability of the market The following part of the code is shown.

```
    function setFeeInfo(address _potion, address _feeToken, uint256 _fe
eAmount, address _feeTo,address _potionFeeTo) external {
        require(hasRole(FEE_SETTER, msg.sender), "!fee setter");
        require(_potion != address(0), "!potion");
        require(_feeToken != address(0), "!feeToken");
        require(_feeTo != address(0), "!feeTo");
        require(_potionFeeTo != address(0),"!potionFeeTo");

        potionToken = IERC20(_potion);
        feeToken = IERC20(_feeToken);
        feeAmount = _feeAmount;
        feeTo = _feeTo;
        potionFeeTo = _potionFeeTo;
    }
```

- **Safety advice**

Timelock contract has been added to the project, it is recommended to keep the administrator's private key properly to ensure the storage security of the private key.

- **Repair Status**

CyberCat has confirmed the risk.

4.3.2 No events added

- **Risk description**

In RewardClaim contract, CatCore contract, CyberCatBlindBox contract, HealthPotion contract, withdraw method, receiveERC20 method, addEnergy method, removeEnergy method, claimXPSFor method and many other methods In order to keep users and administrators informed of contract operation details, it is recommended to add event logging, as shown in the following code.

```
function setFeeAmount(uint256 _feeAmount) external {
    require(hasRole(FEE_SETTER, msg.sender), "!fee setter");
    feeAmount = _feeAmount;
}

function setFeeToPercent(uint256 _feeToPercent) external{
    require(hasRole(FEE_SETTER, msg.sender), "!fee setter");
    require(_feeToPercent <= basePercent,"invalid value");
    feeToPercent = _feeToPercent;
}
```

- **Safety advice**

Suggest adding event logging for sensitive operations.

- **Repair Status**

CyberCat has confirmed the risk.

4.3.3 Claim Logic Analysis

- **Risk description**

RewardClaim contract, claim method, the method appears to be a reward function, but here the incoming msg.value and feeAmount will be judged, after which the amount of feeAmount will be transferred, although the value of the feeAmount variable is controllable by the administrator, but the logic here is not clear, as shown in the following code.

```
function claim() external payable {
    require(msg.value >= feeAmount,"invalid value");
    feeTo.transfer(feeAmount);
    emit RewardClaimed(msg.sender);
}
```

- **Safety advice**

It is recommended that the claim method logic be clarified to avoid the risk of accidental transfers.

- **Repair Status**

CyberCat has confirmed the risk.

4.3.4 breedCat Logic Analysis

- **Risk description**

CatCore contract, breedCat method mainly functions as a breeding cat, the method first determines whether the NFT Token is valid through the modifier, determines whether the NFT of the operation is owned by the caller through require, and determines whether the NFT belonging to the caller is breedable, after which the caller needs to transfer the breeding fee, after which the contract logic will transfer the new NFT Token to the caller address, here the contract logic is in the caller to the transfer, and then transfer the NFT to the caller address, so there is no risk of state changes occurring after the transfer exists, as shown in the following code.

```
    function breedCat(uint256 _sireId, uint256 _matronId)  external ove
rride
            notForbidToken(_sireId)
            notForbidToken(_matronId)
            mustBeValidToken(_sireId)
            mustBeValidToken(_matronId) returns (uint256) {
        require(isBreedable(_sireId, _matronId), "!breedable");
        require(ownerOf(_sireId) == msg.sender && ownerOf(_matronId) ==
 msg.sender, "!owner");
        Cat storage sire = cats[_sireId];
        Cat storage matron = cats[_matronId];
        // deduct fee
        uint256 needPotion = needPotions(sire.breed, matron.breed);
        potionToken.safeTransferFrom(msg.sender, potionFeeTo, needPotio
n);
        uint256 feeToAmount = feeAmount.mul(feeToPercent).div(basePerce
nt);
        uint256 feeToDeadAmount = feeAmount.sub(feeToAmount);
        feeToken.safeTransferFrom(msg.sender, feeTo, feeToAmount);
        feeToken.safeTransferFrom(msg.sender,feeToDead,feeToDeadAmoun
t);
        uint256 newCatId = cats.length;
        cats.push(Cat(0, _sireId, _matronId, 0, msg.sender, now));
        _mint(msg.sender, newCatId);
        sire.breed = sire.breed + 1;
        matron.breed = matron.breed + 1;
        emit CatSpawned(newCatId, msg.sender, 0);
        stateHash = keccak256(abi.encode(_sireId, _matronId, now, state
Hash));
        setTokenURI(newCatId);
        return newCatId;
    }
```

- **Audit Results : Passed**

4.3.5 morphToAdult Logic Analysis

- **Risk description**

CatCore contract, morphToAdult method is the main function of morph into adult. The method determines whether the caller will be the contract address through modifiers, after which it will determine whether the input parameter catid value is reasonable. The method logic first determines whether the NFT of the operation is owned by the caller through require and whether the time satisfies the condition, after which the new gene of the gene is updated, and the logic that can bypass the direct change to adult is not found through analysis, as shown in the following code.

```
    function morphToAdult(uint256 _catId) external override
            notContract()
            mustBeValidToken(_catId) returns (uint256) {
        Cat storage cat = cats[_catId];
        require(cat.gene == 0, "!adult");
        require(now >= cat.bornAt + MatureDays, "!matured");
        require(ownerOf(_catId) == msg.sender, "!owner");

        Cat memory sire = cats[cat.sire];
        Cat memory matron = cats[cat.matron];
        uint256 newGene = geneScience.morphGene(sire.gene, matron.gene,
 stateHash);
        cat.gene = newGene;

        emit CatEvolved(_catId, 0, newGene);

        // update state hash
        stateHash = keccak256(abi.encode(_catId, now, gasleft(), stateH
ash));

        return newGene;
    }
```

- **Audit Results : Passed**

4.3.6 _spawnCat Logic Analysis

- **Risk description**

CatCore contract, _spawnCat method, the method is called by the mintInit method, the above method has not found the logic problem, and mintInit method in the first line of the prompt message is admin setter, that is, the administrator can be set, here you need to strictly review the administrator to avoid the uncontrollable situation.

```
    function mintInit(uint256 _size) external {
         require(hasRole(DEFAULT_ADMIN_ROLE, msg.sender), "!admin sette
r");
         require(InitMinted < InitSupply,"Inited");
         uint256 initLeft = InitSupply - InitMinted;
         if(_size > initLeft){
             _size = initLeft;
         }

          for (uint i = 1; i <= _size; ++i){
            _spawnCat(0, msg.sender);
          }

          InitMinted = InitMinted + _size;
    }

    function _spawnCat(uint256 _gene, address _owner) private returns
(uint256) {
        Cat memory _cat = Cat(_gene, 0, 0, 0, _owner, now);
        uint256 _catId = cats.length;
        cats.push(_cat);
        _mint(_owner, _catId);
        setTokenURI(_catId);

        emit CatSpawned(_catId, _owner, _gene);

        return _catId;
    }
```

- **Audit Results : Passed**

4.3.7 buyByU Logic Analysis

- **Risk description**

CyberCatBlindBox contract, buyByU method is the main function of the user to buy NFT through USDT, here firstly, it will judge whether the user's purchase quantity meets the conditions, after that, the user pays the corresponding fee for the NFT that needs to be bought, after that, it will judge the number of NFT purchased by the user and send them one by one, here the logic is that the user transfers first and then updates the user NFT transfer mode, there is no risk of re-entry, as shown in the following code.

```
    function buyByU(uint256 _collectionId, uint256 _size)  external c
anBuyBox(_collectionId,_size)  {

        Collection storage collection = allCollections[_collectionId];
        require(collection.uSoldCount.add(_size) <= collection.uLimitSi
ze, "Not enough left");
        uint256 cost = uPrice.mul(_size);
        uToken.safeTransferFrom(msg.sender,feeAddress,cost);
        uint256 colId = _collectionId;
        for(uint256 index = 0; index < _size; index++){
            nextIndex = nextIndex.add(1);
            uint256 boxIndex = randomBoxIndex(colId);
            uint256 boxId = boxsByCollectionId[colId][boxIndex];
            uint256 nftId = boxNftMap[boxId];
            NFT storage nft = allNFTs[nftId];
            nft.stay = false;
            IERC721(nftAddress).safeTransferFrom(address(this),msg.send
er,nft.tokenId);
            collection.soldCount = collection.soldCount.add(1);
            collection.uSoldCount = collection.uSoldCount.add(1);
            RequestInfo memory info = RequestInfo(msg.sender,colId,next
Index,boxId,nft.tokenId);
            userBuyBoxMap[msg.sender][colId].push(nextIndex);
            requestMap[nextIndex] = info;
            emit BuyBox(msg.sender,colId,nextIndex,nft.tokenId);
        }
        //emit OpenBox(msg.sender,_collectionId,_index,info.boxId);
    }
```

- **Audit Results : Passed**

4.3.8 Variables are updated

- **Risk description**

When there is a contract logic to obtain rewards or transfer funds, the coder mistakenly updates the value of the variable that sends the funds, so that the user can use the value of the variable that is not updated to obtain funds, thus affecting the normal operation of the project.

- **Audit Results : Passed**

4.3.9 Floating Point and Numeric Precision

- **Risk Description**

In Solidity, the floating-point type is not supported, and the fixed-length floating-point type is not fully supported. The result of the division operation will be rounded off, and if there is a decimal number, the part after the decimal point will be discarded and only the integer part will be taken, for example, dividing 5 pass 2 directly will result in 2. If the result of the operation is less than 1 in the token operation, for example, 4.9 tokens will be approximately equal to 4, bringing a certain degree of The tokens are not only the tokens of the same size, but also the tokens of the same size. Due to the economic properties of tokens, the loss of precision is equivalent to the loss of assets, so this is a cumulative problem in tokens that are frequently traded.

- **Audit Results : Passed**

4.3.10 Default Visibility

- **Risk description**

In Solidity, the visibility of contract functions is public pass default. therefore, functions that do not specify any visibility can be called externally pass the user. This can lead to serious vulnerabilities when developers incorrectly ignore visibility specifiers for functions that should be private, or visibility specifiers that can only be called from within the contract itself. One of the first hacks on Parity’s multi-signature wallet was the failure to set the visibility of a function, which defaults to public, leading to the theft of a large amount of money.

- **Audit Results : Passed**

4.3.11 tx.origin authentication

- **Risk Description**

tx.origin is a global variable in Solidity that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in a smart contract can make the contract vulnerable to phishing-like attacks.

- **Audit Results : Passed**

4.3.12 Faulty constructor

- **Risk description**

Prior to version 0.4.22 in solidity smart contracts, all contracts and constructors had the same name. When writing a contract, if the constructor name and the contract name are not the same, the contract will add a default constructor and the constructor you set up will be treated as a normal function, resulting in your original contract settings not being executed as expected, which can lead to terrible consequences, especially if the constructor is performing a privileged operation.

- **Audit Results : Passed**

4.3.13 Unverified return value

- **Risk description**

Three methods exist in Solidity for sending tokens to an address: transfer(), send(), call.value(). The difference between them is that the transfer function throws an exception throw when sending fails, rolls back the transaction state, and costs 2300gas; the send function returns false when sending fails and costs 2300gas; the call.value method returns false when sending fails and costs all gas to call, which will lead to the risk of reentrant attacks. If the send or call.value method is used in the contract code to send tokens without checking the return value of the method, if an error occurs, the contract will continue to execute the code later, which will lead to the thought result.

- **Audit Results : Passed**

4.3.14 Insecure random numbers

- **Risk Description**

All transactions on the blockchain are deterministic state transition operations with no uncertainty, which ultimately means that there is no source of entropy or randomness within the blockchain ecosystem. Therefore, there is no random number function like rand() in Solidity. Many developers use future block variables such as block hashes, timestamps, block highs and lows or Gas caps to generate random numbers. These quantities are controlled pass the miners who mine them and are therefore not truly random, so using past or present block variables to generate random numbers could lead to a destructive vulnerability.

- **Audit Results : Passed**

4.3.15 Timestamp Dependency

- **Risk description**

In blockchains, data block timestamps (block.timestamp) are used in a variety of applications, such as functions for random numbers, locking funds for a period of time, and conditional statements for various time-related state changes. Miners have the ability to adjust the timestamp as needed, for example block.timestamp or the alias now can be manipulated pass the miner. This can lead to serious vulnerabilities if the wrong block timestamp is used in a smart contract. This may not be necessary if the contract is not particularly concerned with miner manipulation of block timestamps, but care should be taken when developing the contract.

- **Audit Results : Passed**

4.3.16 Transaction order dependency

- **Risk description**

In a blockchain, the miner chooses which transactions from that pool will be included in the block, which is usually determined pass the gasPrice transaction, and the miner will choose the transaction with the highest transaction fee to pack into the block. Since the information about the transactions in the block is publicly available, an attacker can watch the transaction pool for transactions that may contain problematic solutions, modify or revoke the attacker's privileges or change the state of the contract to the attacker's detriment. The attacker can then take data from this transaction and create a higher-level transaction gasPrice and include its transactions in a block before the original, which will preempt the original transaction solution.

- **Audit Results : Passed**

4.3.17 Delegatecall

- **Risk Description**

In Solidity, the delegatecall function is the standard message call method, but the code in the target address runs in the context of the calling contract, i.e., keeping msg.sender and msg.value unchanged. This feature supports implementation libraries, where developers can create reusable code for future contracts. The code in the library itself can be secure and bug-free, but when run in another application's environment, new vulnerabilities may arise, so using the delegatecall function may lead to unexpected code execution.

- **Audit Results : Passed**

4.3.18 Call

- **Risk Description**

The call function is similar to the delegatecall function in that it is an underlying function provided pass Solidity, a smart contract writing language, to interact with external contracts or libraries, but when the call function method is used to handle an external Standard Message Call to a contract, the code runs in the environment of the external contract/function The call function is used to interact with an external contract or library. The use of such functions requires a determination of the security of the call parameters, and caution is recommended. An attacker could easily borrow the identity of the current contract to perform other malicious operations, leading to serious vulnerabilities.

- **Audit Results : Passed**

4.3.19 Denial of Service

- **Risk Description**

Denial of service attacks have a broad category of causes and are designed to keep the user from making the contract work properly for a period of time or permanently in certain situations, including malicious behavior while acting as the recipient of a transaction, artificially increasing the gas required to compute a function causing gas exhaustion (such as controlling the size of variables in a for loop), misuse of access control to access the private component of the contract, in which the Owners with privileges are modified, progress state based on external calls, use of obfuscation and oversight, etc. can lead to denial of service attacks.

- **Audit Results : Passed**

4.3.20 Logic Design Flaw

- **Risk Description**

In smart contracts, developers design special features for their contracts intended to stabilize the market value of tokens or the life of the project and increase the highlight of the project, however, the more complex the system, the more likely it is to have the possibility of errors. It is in these logic and functions that a minor mistake can lead to serious depasstions from the whole logic and expectations, leaving fatal hidden dangers, such as errors in logic judgment, functional implementation and design and so on.

- **Audit Results : Passed**

4.3.21 Fake recharge vulnerability

- **Risk Description**

The success or failure (true or false) status of a token transaction depends on whether an exception is thrown during the execution of the transaction (e.g., using mechanisms such as require/assert/revert/throw). When a user calls the transfer function of a token contract to transfer funds, if the transfer function runs normally without throwing an exception, the transaction will be successful or not, and the status of the transaction will be true. When balances[msg.sender] < _value goes to the else logic and returns false, no exception is thrown, but the transaction acknowledgement is successful, then we believe that a mild if/else judgment is an undisciplined way of coding in sensitive function scenarios like transfer, which will lead to Fake top-up vulnerability in centralized exchanges, centralized wallets, and token contracts.

- **Audit Results : Passed**

4.3.22 Short Address Attack Vulnerability

- **Risk Description**

In Solidity smart contracts, when passing parameters to a smart contract, the parameters are encoded according to the ABI specification. the EVM runs the attacker to send encoded parameters that are shorter than the expected parameter length. For example, when transferring money on an exchange or wallet, you need to send the transfer address address and the transfer amount value. The attacker could send a 19-passte address instead of the standard 20-passte address, in which case the EVM would fill in the 0 at the end of the encoded parameter to make up the expected length, which would result in an overflow of the final transfer amount parameter value, thus changing the original transfer amount.

- **Audit Results : Passed**

4.3.23 Uninitialized storage pointer

- **Risk description**

EVM uses both storage and memory to store variables. Local variables within functions are stored in storage or memory pass default, depending on their type. uninitialized local storage variables could point to other unexpected storage variables in the contract, leading to intentional or unintentional vulnerabilities.

- **Audit Results : Passed**

4.3.24 Frozen Account bypass

- **Risk Description**

In the transfer operation code in the contract, detect the risk that the logical functionality to check the freeze status of the transfer account exists in the contract code and can be passpassed if the transfer account has been frozen.

- **Audit Results : Passed**

4.3.25 Uninitialized

- **Risk description**

The initialize function in the contract can be called pass another attacker before the owner, thus initializing the administrator address.

- **Audit Results : Passed**

4.3.26 Reentry Attack

- **Risk Description**

An attacker constructs a contract containing malicious code at an external address in the Fallback function When the contract sends tokens to this address, it will call the malicious code. The call.value() function in Solidity will consume all the gas he receives when it is used to send tokens, so a re-entry attack will occur when the call to the call.value() function to send tokens occurs before the actual reduction of the sender’s account balance. The re-entry vulnerability led to the famous The DAO attack.

- **Audit Results : Passed**

4.3.27 Integer Overflow

- **Risk Description**

Integer overflows are generally classified as overflows and underflows. The types of integer overflows that occur in smart contracts include three types: multiplicative overflows, additive overflows, and subtractive overflows. In Solidity language, variables support integer types in steps of 8, from uint8 to uint256, and int8 to int256, integers specify fixed size data types and are unsigned, for example, a uint8 type , can only be stored in the range 0 to 2^8-1, that is, [0,255] numbers, a uint256 type can only store numbers in the range 0 to 2^256-1. This means that an integer variable can only have a certain range of numbers represented, and cannot exceed this formulated range. Exceeding the range of values expressed pass the variable type will result in an integer overflow vulnerability.

- **Audit Results : Passed**

5. Security Audit Tool

Tool name	Tool Features
Oyente	Can be used to detect common bugs in smart contracts
securify	Common types of smart contracts that can be verified
MAIAN	Multiple smart contract vulnerabilities can be found and classified
Lunaray Toolkit	self-developed toolkit

Disclaimer：

Lunaray Technology only issues a report and assumes corresponding responsibilities for the facts that occurred or existed before the issuance of this report, Since the facts that occurred after the issuance of the report cannot determine the security status of the smart contract, it is not responsible for this.

Lunaray Technology conducts security audits on the security audit items in the project agreement, and is not responsible for the project background and other circumstances, The subsequent on-chain deployment and operation methods of the project party are beyond the scope of this audit.

This report only conducts a security audit based on the information provided by the information provider to Lunaray at the time the report is issued, If the information of this project is concealed or the situation reflected is inconsistent with the actual situation, Lunaray Technology shall not be liable for any losses and adverse effects caused thereby.

There are risks in the market, and investment needs to be cautious. This report only conducts security audits and results announcements on smart contract codes, and does not make investment recommendations and basis.

LUNARAY

BLOCKCHAINSECURITY

https://lunaray.co

https://github.com/lunaraySec

https://twitter.com/lunaray_Sec

http://t.me/lunaraySec