LUNARAY
BLOCKCHAINSECURITY

SMART CONTRACT SECURITY AUDIT REPORT

For W3Bridge

16 May 2023

lunaray.co

Table of Contents

1. Overview

On May 4, 2023, the security team of Lunaray Technology received the security audit request of the **W3BRIDGE project**. The team completed the audit of the **W3BRIDGE smart contract** on May 16, 2023. During the audit process, the security audit experts of Lunaray Technology and the W3BRIDGE 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 W3BRIDGE 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 W3BRIDGE smart contract security audit: **Passed**

Audit Report Hash:
AF4546926D6CD39F0D0E10763D88C32535A5F4E28CF3A6A3E4BCB5C907ECDA98

2. Background

2.1 Project Description

Project name	W3Bridge
Contract type	Span Chain Bridge
Code language	Solidity
Contract file	ChainDelegater.sol,ExchangePair.sol,ExchangePermission.sol,HostChain.sol,WrappedCoin.sol,Migrations.sol,SyncChain.sol,ExchangeFactory.sol,ExchangePairBeacon.sol,InnerChain.sol,ExchangeRouter.sol
Brief introduction	The W3Bridge project is a solution to the inevitable cross-chain interaction problem in the current blockchain ecosystem. It aims to realize cross-chain interaction between different blockchains and provide an efficient and reliable cross-chain bridge to allow digital assets to be transferred between different blockchain networks without barriers. The project adopts on-chain synchronization chain to realize cross-chain bridging by creating smart contracts on heterogeneous chains and performing operations such as event listening and status query to realize the transmission and parsing of information on another chain.

2.2 Audit Range

Smart contract file name and corresponding SHA256：

Name	SHA256
ChainDelegater.sol	52FAFEC43438161F9DFDEBFAE47E499FA2A7006AAF5A6BC39277661990157251
ExchangePair.sol	E586524F5136A0810563220A5294C198D254A00E3B332E5D61C6DBA3369BC7DB
ExchangePermission.sol	84FE284815470F0716CFC8264B06E82E11ADA7D1657CB57C88B5A538318DEDE3
HostChain.sol	7BBD9651CF9DA29D211C5E00F88D6870AAF3E7BBC0206FA5FF3A68B7465DEB55
WrappedCoin.sol	A0F589BE95A0AF96E3B6FE511CD690193C35A6318678FC1D74BC9F2CD4CD51B5
Migrations.sol	A1DD9B5B134B8F983AAF46C8C317BD21774A5C25129F7C9E4A53BA544CBC7C8D
SyncChain.sol	2C34BFDA1BFB23AB83C703DFE094D98F67A7FB2CE8EEB1A8FF85BC76F404914C
ExchangeFactory.sol	AF8EB93B7C60FE8189183A3751D4326F32F40D70C40F057B750D2FD4D10CBE99
ExchangePairBeacon.sol	1E1C305B27C60EF0A52D019A77AE13761B6763581A2AF09F8EF4B9CE3546028E
InnerChain.sol	584BD830AF08BCA92AA03D21CD504EF5D3676CED00375E7AFD5564208B8F5221

ExchangeRouter.sol	76445515F4C842E2F0C133338EA32098A91DFAD1703FCCC44B0070B68CC9495F

3. Project contract details

3.1 Contract Overview

ChainDelegater Contract

The contract uses Initializable and AccessControlUpgradeable from the OpenZeppelin library to initialize the contract and configure permissions, providing the administrator with configuration of the synchronous chain contract address and fees. It also provides the ability to return configuration information from the previous epoch, as well as a public function to transfer funds within the contract, and a transfer of funds out of the contract that can only be called by the administrator.

ExchangeFactory Contract

This contract is derived from the ExchangePermission contract and implements permission management and initialization operations. The main logic of the contract is to implement a cross-chain transaction pair factory contract to create cross-chain transaction pairs and set parameters such as fees. The contract supports designated administrators to create pairs between various ERC20 tokens and WCOIN (the chain's native token), as well as to add or remove tokens that can be exchanged with the specified ERC20 tokens, and to set per-transaction fees, and additional fees. Implemented public viewing of on-chain data, including viewing available cross-chain tokens and querying cross-chain fees.

ExchangePair Contract

The contract is derived from the Initializable contract to implement the contract initialization operation, and a series of initialization parameters are defined in the contract, such as whether it is a contract that supports mint destruction and the corresponding initialization operation, the factory contract address of the transaction pair, and the cross-chain token configuration. The contract supports users to deposit money into the contract and withdraw funds from the contract. The deposited funds can be operated across chains, and funds from other chains across chains can be withdrawn. The smart contract can also add and remove token information needed for cross-chain transactions, but this function is called indirectly through the administrator of the factory contract. Two internal functions _beforExchangeFromProof and _afterExchangeToChain are implemented to submit cross-chain transactions, and to validate and use cross-chain transaction proofs. The smart contract implements two cross-chain functions exchangeToUseBorrowAmount, exchangeTo, the first of which supports cross-chaining using funds deposited by the user, and the second of which supports the user to transfer funds again when doing so. The smart contract also implements a function to withdraw assets in bulk and add them to the lending pool via proof of transaction, while rewarding the caller with points. In addition, it implements a function that allows users to query the status of the corresponding cross-chain proofs by chain id, submitter and proof id. This smart contract can prevent re-entry attacks by users by using a re-entry function and a checksum lock. Overall the contract implements a function that can initiate cross-chain transactions, as well as validate cross-chain transactions, it can handle cross-chain fees, validate transaction certificates, and handle user redemption and redemption requests, and record the status of the transactions.

ExchangePairBeacon Contract

The contract mainly implements a factory contract (BeaconProxy) of a scalable proxy contract as a standard ERC1967 EIP that can upgrade the contract without migrating data. The ExchangePair contract is used to match and trade cross-chain pairs. When the contract is deployed, it automatically instantiates an ExchangePair contract and provides an external access interface to ExchangePair. Users can access this interface to get information about the transaction pairs.

ExchangePermission Contract

The contract mainly implements a factory contract (BeaconProxy) of a scalable proxy contract as a standard ERC1967 EIP that can upgrade the contract without migrating data. The ExchangePair contract is used to match and trade cross-chain pairs. When the contract is deployed, it automatically instantiates an ExchangePair contract and provides an external access interface to ExchangePair. Users can access this interface to get information about the transaction pairs.

ExchangeRouter Contract

The contract is a cross-chain transaction contract that enables transfer from one token to another, while supporting on-chain and off-chain token exchange. On initialization, the binding factory contract and the chain-native token ERC20 are bound to the contract. Two functions are implemented internally, where the acceptExchangeTokens function looks at the list of accepted cross-chain tokens and the tokenExchangeToChain function performs the action of transferring assets from the chain to other chains. The internal use of call factory contract, call token contract and cross-chain transaction pair contract.

HostChain Contract

The smart contract is derived from the InnerChain contract which implements the basic synchronization chain functionality, initializing the smart contract with the initialize function, which gives the initial state of the block chain. The contract provides several callable functions, including getBlockTransactionBreakPoint, getTransactionHeadsByBlockHash, and getTransactionHeadsByBlockNumber, which are used to obtain information such as the transactions within a given block. In addition, the consensus parameters are updated at each new epoch, and the submitBlockHead function implements the block submission function.

InnerChain Contract

The contract is an abstract contract for implementing the basic functions of the synchronization chain. It has the ability to synchronize block headers and epoch configurations. The contract also provides a number of functions that can be called to obtain information such as blocks and transactions. The basic contract operations include initialization, consensus checksum, fetching block header, fetching epoch configuration, estimating fees, submitting transactions to the chain, and transaction validation. The implementation of the smart contract is mainly based on OpenZeppelin's libraries and interfaces, while some custom libraries are used to implement a unified blockchain data structure, which includes Chain structure, BlcokHead structure, TransactionHead structure, etc.

Migrations Contract

This contract implements a state that manages a certain migration. It allows only the creator of the contract to call the setCompleted function to update the number of the most recently completed migration. This ensures that only authorized people can update the status, and that the contract can be used to record when a migration is indicated as completed.

Migrations Contract

The contract inherits the InnerChain contract to implement basic synchronization chain functionality, mainly for synchronizing block headers and epoch configurations. Among other things, the contract includes EpochConfigLib, ChainLib, BlockHeadLib and Consensus libraries. The contract constructor initialize is used to initialize the contract identifier, epoch configuration, genesis block and chain delegate information. syncBlockHead function is used to synchronize the block header and signature of the current chain in bulk, perform data verification and weight verification, and add the block to the chain if the verification passes. syncEpochConfig function is used to synchronize the epoch The syncEpochConfig function is used to synchronize epoch configuration information and signatures, verify that the signer is the sponsor and that the signature is valid, and add epoch configuration information to the epochConfigOf mapping table after consensus is reached. whenAppendNewBlock function is called as an optional implementation after the block reaches consensus.

WrappedCoin Contract

This contract is a bridge contract between ERC20 Token and chain native tokens, mainly responsible for binding ERC20 Token of native chain tokens, and realizing the conversion of native token tokens to WrappedCoin Token or WrappedCoin Token to native tokens. It also encapsulates the ERC20 Token standard function interface. The contract does not have any administrator operations, and the funds stored by the user can only be withdrawn by the user.

3.2 Contract details

ChainDelegater Contract

Name	Parameter	Attributes
initialize	none	public
whenEnterNewEpochDelegate	uint64 epochIndex	external
whenAppendNewBlockDelegate	BlockHead innerBlock	external
setChainImplement	IChain _chain	onlyRole
setFeeConfig	FeeConfig config	onlyRole
whenEmitedL3Transaction	address emitter uint256 nonce	external
withdrawFeeTo	addresspayable to uint256 amount	onlyRole

ExchangeFactory Contract

Name	Parameter	Attributes
initialize	address _l3 address _pairBeacon address _wcoin	public
_requirePair	address token	internal
fee	none	external
_createExchangePair	address tokenContractAddress bool supportMintBurnInterface	onlyRole
createExchangePair	address tokenContractAddress	onlyRole
createExchangePairSupportPairMintBurnInterface	address tokenContractAddress	onlyRole
setFee	Fee memory feeConfig	onlyRole

setFeeAdditional	address pair FeeAdditional memory feeAdditional	onlyRole
acceptExchangeTokens	address token	external
addAcceptExchangeTokenId	address targetToken ExchangeTokenId memory etid	onlyRole
delAcceptExchangeTokenId	address targetToken ExchangeTokenId memory etid	onlyRole
feeAdditionalOf	address pair	external

ExchangePair Contract

Name	Parameter	Attributes
initialize	address _token bytes32 _chainIdentifier bool _isMintBurnPair	external
_handleFees	uint256 originAmount	internal
_beforExchangeFromProof	TransactionProof proof ExchangeCertificate cert	internal
_afterExchangeToChain	ExchangeTokenId toETID address toAccount uint256 amount uint256 feeAmount uint256 feeAdditionalAmount	internal
acceptExchangeTokenIds	none	external
addAcceptExchangeTokenId	ExchangeTokenId etid	external
delAcceptExchangeTokenId	ExchangeTokenId etid	external

withdrawBorrowAmount	address toAccount uint256 amount	external
depositBorrowAmount	uint256 amount	external
exchangeToUseBorrowAmount	ExchangeTokenId toETID address toAccount uint256 amount	external
sync	none	public
exchangeToEstimateFee	ExchangeTokenId toETID address fromAccount address toAccount uint256 amount	external
exchangeTo	ExchangeTokenId toETID address fromAccount address toAccount	external
exchangeFromProofsWithAddLiquidity	TransactionProof[] memory proofs bool ignoreRewards	external
exchangeFromProof	TransactionProof proof	external
_safeWithdrawTransfer	address to uint256 amount	internal
certificateStateOf	bytes32 chainIdentifier address shadownEmiter uint256 certificateId	external

ExchangePermission Contract

Name	Parameter	Attributes
__ExchangePermission_Init	none	onlyInitializing

ExchangeRouter Contract

Name	Parameter	Attributes
initialize	address _factory	public
_requirePair	address token	internal
acceptExchangeTokens	address token	external
tokenExchangeToChainEstimateFee	ExchangeTokenId fromETID ExchangeTokenId toETID address toAccount uint256 amount	external
coinExchangeToChain	ExchangeTokenId coinETID ExchangeTokenId toETID address toAccount	external
tokenExchangeToChain	ExchangeTokenId fromETID ExchangeTokenId toETID address toAccount uint256 amount	external

HostChain Contract

Name	Parameter	Attributes
initialize	bytes32 identifier address[] memory initVerifiers uint128 blockSize uint16 blockInterval IChainDelegate chainDelegate	public
getBlockTransactionBreakPoint	bytes32 blockHash	external
getTransactionHeadsByBlockHash	bytes32 hash	external
getTransactionHeadsByBlockNumber	uint64 number	external
epochUpdate	none	external

submitBlockHead	BlockHead memory innerBlock TransactionHead[] memory transactions bytes memory signature	external

InnerChain Contract

Name	Parameter	Attributes
__InnerChain_init	bytes32 identifier EpochConfig genesisEpochConfig BlockHead genesisBlockHead IChainDelegate chainDelegate	onlyInitializing
genesisBlock	none	external
getEpochConfigAtIndex	uint64 epochIndex	external
_proposalBlock	EpochConfig config BlockHead innerBlock bytes signature	internal
getBlockNumber	none	external
getBlockHeadByHash	bytes32 blockHash	external
getBlockHeadByNumber	uint64 number	external
isAgreedProposals	uint64[] memory blockNumbers bytes32[] memory blockHashs address proposal	external
setFeeConfig	FeeConfig config	external
estimateFee	uint256 len	external
send	bytes memory datas	external
verify	TransactionProof proof	external

Migrations Contract

Name	Parameter	Attributes
setCompleted	uint256 completed	public

SyncChain Contract

Name	Parameter	Attributes
initialize	bytes32 identifier EpochConfig epochConfig BlockHead genesisBlock IChainDelegate chainDelegate	public
syncBlockHead	BlockHead[] memory innerBlock bytes memory signature	external
syncEpochConfig	EpochConfig config bytes signature	external
whenAppendNewBlock	BlockHead innerBlock	internal

WrappedCoin Contract

Name	Parameter	Attributes
deposit	none	external
depositTo	address to	external
withdraw	uint256 amount	external
withdrawTo	address payable to uint256 amount	external

4. Audit details

4.1 Findings Summary

Severity	Found	Resolved	Acknowledged
High	**0**	**0**	0
Medium	0	0	0
Low	1	0	1
Info	1	0	1

4.2 Risk distribution

Name	Risk level	Repair status
Administrator Permissions	Low	Acknowledged
Transaction order dependency	Info	Acknowledged
Logical Design Flaw	No	normal
Reentry attack	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
Delegatecall	No	normal
Call	No	normal
Denial of Service	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
Integer Overflow	No	normal

4.3 Risk audit details

4.3.1 Administrator Permissions

- **Risk description**

When the administrator address is an EOA address, there is a risk that the private key will be compromised by malicious phishing or other means, resulting in malicious modification of the project's administrator-only configuration.

```
contract ExchangeFactory is ExchangePermission, IExchangeFactory {
    ...
    function _createExchangePair(
        address tokenContractAddress,
        bool supportMintBurnInterface
    ) private onlyRole(MANAGER_ROLE) {
        require(
            pairOf[tokenContractAddress] == IExchangePair(address(0)),
            "EFactory:PAIR_EXISTED"
        );
        pairOf[tokenContractAddress] = IExchangePair(
            address(
                new BeaconProxy(
                    address(pairBeacon),
                    abi.encodeWithSignature(
                        "initialize(address,bytes32,bool)",
                        tokenContractAddress,
                        l3.chainIdentifier(),
                        supportMintBurnInterface
                    )
                )
            )
        );
        emit CreatedExchangePair({
            chainIdentifier: l3.chainIdentifier(),
            tokenContract: tokenContractAddress,
            tokenName: IERC20MetadataUpgradeable(tokenContractAddress).
name(),
            tokenSymbol: IERC20MetadataUpgradeable(tokenContractAddres
s)
                .symbol(),
            tokenDecimals: IERC20MetadataUpgradeable(tokenContractAddre
ss)
```

```
                .decimals(),
            pairContract: address(pairOf[tokenContractAddress])
        });
    }
    function createExchangePair(
        address tokenContractAddress
    ) external onlyRole(MANAGER_ROLE) {
        _createExchangePair(tokenContractAddress, false);
    }
    function createExchangePairSupportPairMintBurnInterface(
        address tokenContractAddress
    ) external onlyRole(MANAGER_ROLE) {
        _createExchangePair(tokenContractAddress, true);
    }
    function setFeeAdditional(
        address pair,
        FeeAdditional memory feeAdditional
    ) external onlyRole(FEE_MANAGER_ROLE) {
        require(feeAdditional.feeTo != address(0), "EFactory:INVAILD_FE
ETO");
        require(feeAdditional.rate <= 1e12, "EFactory:INVAILD_RATE");
        _feeAdditionalOf[pair] = feeAdditional;
    }
    function addAcceptExchangeTokenId(
        address targetToken,
        ExchangeTokenId memory etid
    ) external onlyRole(MANAGER_ROLE) {
        IExchangePair pair = _requirePair(targetToken);
        pair.addAcceptExchangeTokenId(etid);
    }
    function delAcceptExchangeTokenId(
        address targetToken,
        ExchangeTokenId memory etid
    ) external onlyRole(MANAGER_ROLE) {
        IExchangePair pair = _requirePair(targetToken);
        pair.delAcceptExchangeTokenId(etid);
    }
    ...
}

contract ChainDelegater is
    Initializable,
    AccessControlUpgradeable,
    IChainDelegate
{
    ...
```

```
    function setChainImplement(
        IChain _chain
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        chain = _chain;
    }
    function setFeeConfig(
        FeeConfig memory config
    ) external onlyRole(MANAGER_ROLE) {
        chain.setFeeConfig(config);
    }

    ...

    function withdrawFeeTo(
        address payable to,
        uint256 amount
    ) external onlyRole(MANAGER_ROLE) {
        to.transfer(amount);
    }
}
```

- **Safety advice**

It is recommended to use multi-signature contracts or multi-signature wallets to con trol administrator permissions, or to destroy administrator permissions after contra ct chaining.

- **Repair Status**

W3Bridge has Acknowledged. The project owner guarantees that EOA administrator privileges will be transferred to the multi-signature wallet once the project is deployed and configured.

4.3.2 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.

This function uses an external transfer of ERC20 Token and then calculates the balance difference. When the transfer transaction is not the same transaction as this function, it could be front-loaded by an attacker using a higher fee, which is not a risk since this function is only called in the ExchangeRouter contract in the entire project. However, because the exchangeTo function is external and does not have any calling privileges blocked, this risk can occur when an external user calls the exchangeTo function in the ExchangePair contract alone. This risk has no impact on the project owner and is only relevant to the user who initiated the call to the function.

```
function exchangeTo(
    ExchangeTokenId memory toETID,
    address fromAccount,
    address toAccount
) external payable nonReentrant {
    // unsupport exchange to self chain
    require(
        toETID.chainIdentifier != ETID.chainIdentifier,
        "EPair:NO_EXCHANGE_TO_SELFCHAIN"
    );
    // Processing a transaction based on the actual amount of tokens
received.
    uint256 amount = IERC20MetadataUpgradeable(ETID.tokenContract)
```

```
        .balanceOf(address(this)) - tokenBalnaceReserver;
    require(amount > 0, "EPair:NO_TRANSFER_BEFOR_EXCHANGE");
    // If the token is a self-managed token, it will be burned
directly.
    if (isMintBurnPair) {

IExchangePairMintBurn(ETID.tokenContract).exchangePairBurn(amount);
    } else {
        tokenBalnaceReserver += amount;
    }
    (
        uint256 fainalAmount,
        uint256 feeAmount,
        uint256 feeAdditionalAmount
    ) = _handleFees(amount);
    ExchangeCertificate memory cert = _afterExchangeToChain(
        toETID,
        toAccount,
        fainalAmount,
        feeAmount,
        feeAdditionalAmount
    );
    emit Exchanged({
        fromAccount: fromAccount,
        toAccount: toAccount,
        fromETIDHash: ETID.hash(),
        toETIDHash: toETID.hash(),
        amount: fainalAmount,
        fee: feeAmount,
        feeAdditional: feeAdditionalAmount,
        assetProvider: fromAccount,
        certificateId: cert.cerificateID
    });
}
```

- **Safety advice**

It is recommended that sub-functions be disabled from being called by any user, and that only major business contracts be allowed to make calls.

- **Repair Status**

W3Bridge has Acknowledged.

4.3.3 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**

4.3.4 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**

4.3.5 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**

4.3.6 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**

4.3.7 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**

4.3.8 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**

4.3.9 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**

4.3.10 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**

4.3.11 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**

4.3.12 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**

4.3.13 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**

4.3.14 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**

4.3.15 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**

4.3.16 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**

4.3.17 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**

4.3.18 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**

4.3.19 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**

4.3.20 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**

4.3.21 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.

https://lunaray.co

https://github.com/lunaraySec

https://twitter.com/lunaray_Sec

http://t.me/lunaraySec