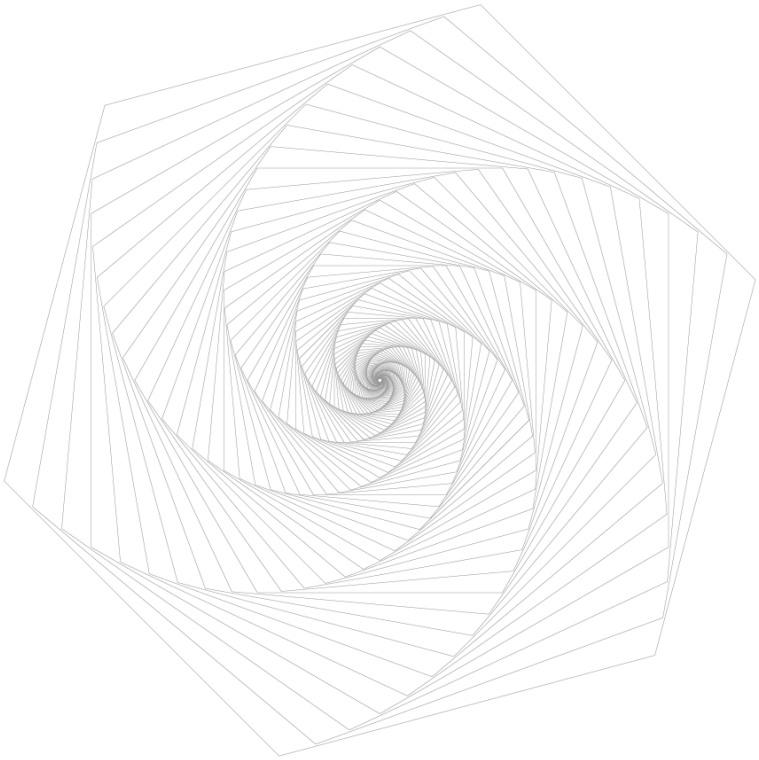
KNOWNSEC
Blockchain Lab

Smart Contract Audit Report



Version description

The revision	Date	Revised	Version
Write documentation	20220125	KNOWNSEC Blockchain Lab	V1.0

Document information

Title	Version	Document Number	Type
HYDRA Smart Contract Audit Report	V1.0	81581cc445cb49aba05f03285cc883fa	Open to project team

Statement

KNOWNSEC Blockchain Lab only issues this report for facts that have occurred or existed before the issuance of this report, and assumes corresponding responsibilities for this. KNOWNSEC Blockchain Lab is unable to determine the security status of its smart contracts and is not responsible for the facts that will occur or exist in the future. The security audit analysis and other content made in this report are only based on the documents and information provided to us by the information provider as of the time this report is issued. KNOWNSEC Blockchain Lab 's assumption: There is no missing, tampered, deleted or concealed information. If the information provided is missing, tampered with, deleted, concealed or reflected in the actual situation, KNOWNSEC Blockchain Lab shall not be liable for any losses and adverse effects caused thereby.

Directory

1. Summarize

The effective test period of this report is from **January 19, 2022 to January 25, 2022**. During this period, the security and standardization of **the fund pool code and cross-chain code of the HYDRA smart contract** will be audited and used as a Report statistics.

The scope of this smart contract security audit does not include external contract calls, new attack methods that may appear in the future, and code after contract upgrades or tampering. (With the development of the project, the smart contract may add a new pool , New functional modules, new external contract calls, etc.), does not include front-end security and server security.

In this audit report, engineers conducted a comprehensive analysis of the common vulnerabilities of smart contracts (Chapter 6). **The smart contract code of the HYDRA** is comprehensively assessed as **PASS**.

Since the testing is under non-production environment, all codes are the latest version. In addition, the testing process is communicated with the relevant engineer, and testing operations are carried out under the controllable operational risk to avoid production during the testing process, such as: Operational risk, code security risk.

KNOWNSEC Attest information:

classification	information
report number	81581cc445cb49aba05f03285cc883fa
report query link	https://attest.im/attestation/searchResult?qurey=81581cc445cb49aba05f03285cc883fa

2. Item information

2.1. Item description

HYDRA is an open source proof-of-stake blockchain with a unique set of economic characteristics. It contains a unique combination of inflationary and deflationary mechanisms that work in parallel as part of its economy and let true adoption define its total supply.

2.2. The project's website

https://hydrachain.org/

2.3. White Paper

https://hydrachain.org/

2.4. Review version code

Public（ETH）：

https://cn.etherscan.com/address/0xd70c3a1ab3789e5017bfc09bbd278946d3eb478b#code

Private（Github）：

https://github.com/Hydra-Chain/hydra-bridge

Private chain

Hydra Bridge new - Contract Summary:

ContractAddress : b76db3c7d94e783a302479ab6dc1ae0774fb015d

hERC20 Vault - Contract Summary:

ContractAddress : 8bbd98651b64afbdb288415a1ad9c6a05427951c

hDAI Token - Contract Summary:

ContractAddress : aee4289ec95e04114f2456fc4b16130965b0e114

hETH Token - Contract Summary:

ContractAddress : 026313b614feb50a939c2582ff83549b8077d6f8

2.5. Contract file and Hash/contract deployment address

The contract documents	MD5
HydraBridge.sol	f9e29c5b6b3f2e97a6bcde5e00f4d78d
BaseVault.sol	93b9f9194b34ae8686f739a4a0ac8635
ERC20DefaultVault.sol	0a8d1261ab1d9d9691bd093d5b503ffa
HydraAccessControl.sol	f3fdf6c260262f2c0a72205852f1fd75
WETHVault.sol	5040469a38f55f4b516839b583634827

3. External visibility analysis

3.1. HydraBridge.sol contracts

HydraBridge					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
hasVotedOnProposal	public	False	---	---	---
isObserver	external	False	---	---	---
getProposal	external	False	---	---	---
totalObservers	external	False	---	---	---
_pause	external	True	onlyAdmin	---	---
_unpause	external	True	onlyAdmin	---	---
_setFee	external	True	onlyFeeAdmin	---	---
_changeVoteTreshold	external	True	onlyAdmin	---	---
addObserver	external	True	onlyAdmin	---	---
removeObserver	external	True	onlyAdmin	---	---
_setAssetForVault	external	True	onlyAdmin	---	---
_setAssetBurnable	external	True	onlyAdmin	---	---
_deposit	external	True	whenNotPaused	payable	---
_vote	external	True	onlyObservers	---	---

			whenNotPaused		
execute	external	True	onlyObservers whenNotPaused	---	---
hasVoted	private	False	---	---	---
claimFees	external	True	onlyFeeAdmin	---	---
observerBit	private	False	---	---	---

3.2. BaseVault.sol contracts

BaseVault					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
setAsset	external	True	onlyBridge	---	---
setBurnable	external	True	onlyBridge	---	---
_setAsset	Internal	True	---	---	---
_setBurnable	Internal	True	---	---	---

3.3. ERC20DefaultVault.sol contracts

ERC20DefaultVault					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
lock	external	True	onlyBridge	payable	---
execute	external	True	onlyBridge	---	---

3.4. HydraAccessControl.sol contracts

HydraAccessControl					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
supportsInterface	Public	False	onlyBridge	---	---
getRoleMember	Public	False	onlyBridge	---	---
getRoleMemberCount	Public	False	---	---	---
getRoleMemberIndex	Public	False	---	---	---
grantRole	Public	True	---	---	---
revokeRole	Public	True	---	---	---
renounceRole	Public	True	---	---	---
_setupRole	Internal	True	---	---	---

3.5. WETHVault.sol contracts

WETHVault					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
getLockRecord	external	False	---	---	---
lock	external	True	onlyBridge	payable	---
execute	external	True	onlyBridge	---	---

4. Code vulnerability analysis

4.1. Summary description of the audit results

Audit results			
audit project	**audit content**	**condition**	**description**
Business security detection	ERC20Vault pool lock function	Pass	After testing, there is no security issue.
	ERC20Vault pool execute function	Pass	After testing, there is no security issue.
	WETHVault pool lock and execute functions	Pass	After testing, there is no security issue.
	HydraBridge contract deposit function	Pass	After testing, there is no security issue.
	HydraBridge contract execute function	Pass	After testing, there is no security issue.
Code basic vulnerability detection	Compiler version security	Pass	After testing, there is no security issue.
	Redundant code	Pass	After testing, there is no security issue.
	Use of safe arithmetic library	Pass	After testing, there is no security issue.
	Not recommended encoding	Pass	After testing, there is no security issue.
	Reasonable use of require/assert	Pass	After testing, there is no security issue.
	fallback function safety	Pass	After testing, there is no security issue.

	tx.origin authentication	Pass	After testing, there is no security issue.
	Owner permission control	Pass	After testing, there is no security issue.
	Gas consumption detection	Pass	After testing, there is no security issue.
	call injection attack	Pass	After testing, there is no security issue.
	Low-level function safety	Pass	After testing, there is no security issue.
	Vulnerability of additional token issuance	Pass	After testing, there is no security issue.
	Access control defect detection	Pass	After testing, there is no security issue.
	Numerical overflow detection	Pass	After testing, there is no security issue.
	Arithmetic accuracy error	Pass	After testing, there is no security issue.
	Wrong use of random number detection	Pass	After testing, there is no security issue.
	Unsafe interface use	Pass	After testing, there is no security issue.
	Variable coverage	Pass	After testing, there is no security issue.
	Uninitialized storage pointer	Pass	After testing, there is no security issue.
	Return value call verification	Pass	After testing, there is no security issue.

	Transaction order dependency detection	Pass	After testing, there is no security issue.
	Timestamp dependent attack	Pass	After testing, there is no security issue.
	Denial of service attack detection	Pass	After testing, there is no security issue.
	Fake recharge vulnerability detection	Pass	After testing, there is no security issue.
	Reentry attack detection	Pass	After testing, there is no security issue.
	Replay attack detection	Pass	After testing, there is no security issue.
	Rearrangement attack detection	Pass	After testing, there is no security issue.

5. Business security detection

5.1. ERC20Vault pool lock function【Pass】

Audit analysis: When the cross-chain bridge calls the lock function, the number, type, and recipient address of the cross-chain token are retrieved through the _data data passed in by the user, and the type of the cross-chain token must conform to the tokens in the allowlist. Then judge whether the cross-chain token is in the tokenBurnList, process the cross-chain token according to the situation, and finally record the information in lockRecords. After the audit, the authority control is correct and the logic design is reasonable.

```
function lock(
        bytes32 _assetId,
        uint8 _destinationChainId,
        uint64 _lockNonce,
        address _user,
        bytes calldata _data
    ) external payable override onlyBridge {// knownsec// lock only onlyBridge call
        bytes memory recipientAddress;

        (uint256 amount, uint256 recipientAddressLength) = abi.decode(_data, (uint256, uint256));
        recipientAddress = bytes(_data[64:64 + recipientAddressLength]);

        address tokenAddress = assetIdToTokenAddress[_assetId];
        require(tokenAllowlist[tokenAddress], "Vault: token is not in the allowlist");

        if (tokenBurnList[tokenAddress]) {
            // burn on destination chain
            ERC20Burnable(tokenAddress).burnFrom(_user, amount);// knownsec // burn
```

```
tokens
            } else {
                // lock on source chain
                ERC20Burnable(tokenAddress).safeTransferFrom(_user, address(this), amount);//
knownsec // Staging token contract
            }

            lockRecords[_destinationChainId][_lockNonce] = LockRecord(// knownsec // record
lock to mapping
                tokenAddress,
                _destinationChainId,
                _assetId,
                recipientAddress,
                _user,
                amount
            );
        }
```

Security advice: None.

5.2. ERC20Vault pool execute function【Pass】

Audit analysis: When the cross-chain bridge calls the execute function, the number, type and recipient address of the cross-chain token are firstly extracted through the _data data passed in by the user, and the type of the cross-chain token must conform to the tokens in the allowlist, and finally receive it. The user mints or transfers coins to complete the cross-chain. After the audit, the authority control is correct and the logic design is reasonable.

```
function execute(bytes32 _assetId, bytes calldata _data) external override onlyBridge {// knownsec
```

```
// Transfer out tokens, only called by onlyBridge
        bytes memory destinationRecipientAddress;

        (uint256 amount, uint256 lenDestinationRecipientAddress) = abi.decode(_data,
(uint256, uint256));
        destinationRecipientAddress = bytes(_data[64:64 + lenDestinationRecipientAddress]);

        bytes20 recipientAddress;
        address tokenAddress = assetIdToTokenAddress[_assetId];

        // solhint-disable-next-line
        assembly {
            recipientAddress := mload(add(destinationRecipientAddress, 0x20))
        }

        require(tokenAllowlist[tokenAddress], "Vault: token is not in the allowlist");

        if (tokenBurnList[tokenAddress]) {
            // mint on destination chain
            ERC20PresetMinterPauser(tokenAddress).mint(address(recipientAddress),
amount);
        } else {
            // release on source chain
            ERC20Burnable(tokenAddress).safeTransfer(address(recipientAddress), amount);
        }
    }
```

Security advice: None.

5.3. WETHVault pool lock and execute functions【Pass】

Audit analysis: The functions of lock and execute in the WETHVault pool are similar to those of the ERC20Vault pool, which belong to a single-currency cross-chain

pool. The calling permission of this method is: onlyBridge. After the audit, the authority control is correct and the logic design is reasonable.

```
function lock(
        bytes32 _assetId,
        uint8 _destinationChainId,
        uint64 _lockNonce,
        address _user,
        bytes calldata _data
    ) external payable override onlyBridge {
        bytes memory recipientAddress;

        (uint256 amount, uint256 recipientAddressLength) = abi.decode(_data, (uint256, uint256));
        recipientAddress = bytes(_data[64:64 + recipientAddressLength]);

        require(amount == msg.value, "Vault: value send doesn't match data");

        address tokenAddress = assetIdToTokenAddress[_assetId];
        require(tokenAllowlist[tokenAddress], "Vault: token is not in the allowlist");

        IWETH9(WETH).deposit{ value: amount }();

        lockRecords[_destinationChainId][_lockNonce] = LockRecord(
            tokenAddress,
            _destinationChainId,
            _assetId,
            recipientAddress,
            _user,
            amount
        );
    }
```

```
    function execute(bytes32 _assetId, bytes calldata _data) external override onlyBridge {
        bytes memory destinationRecipientAddress;

        (uint256 amount, uint256 lenDestinationRecipientAddress) = abi.decode(_data, (uint256, uint256));
        destinationRecipientAddress = bytes(_data[64:64 + lenDestinationRecipientAddress]);

        bytes20 recipient;
        address tokenAddress = assetIdToTokenAddress[_assetId];

        // solhint-disable-next-line
        assembly {
            recipient := mload(add(destinationRecipientAddress, 0x20))
        }

        require(tokenAllowlist[tokenAddress], "Vault: token is not in the allowlist");

        IWETH9(WETH).withdraw(amount);
        payable(address(recipient)).transfer(amount);
    }
```

Security advice: None.

5.4. HydraBridge contract deposit function【Pass】

Audit analysis: Conduct security audit on the deposit function of the HydraBridge contract. Its functional logic is: to judge that the user calls this function with a value greater than feeAmount, verify that the vault address is not 0, add the number of pledges, and finally call the vault lock function to complete the deposit function . After the audit, the authority control is correct and the logic design is reasonable.

```
        function deposit(
```

```
            uint8 _destinationChainId,
            bytes32 _assetId,
            bytes calldata data
        ) external payable whenNotPaused {
            uint256 _fee = feeAmount;
            require(msg.value >= _fee, "HydraBridge: fee insufficient");
            address vaultAddress = assetIdToVault[_assetId];
            require(vaultAddress != address(0), "HydraBridge: vault not found");

            uint64 nonce = ++lockNonce[_destinationChainId];

            IVault vault = IVault(vaultAddress);
            vault.lock{ value: msg.value - _fee }(_assetId, _destinationChainId, nonce,
msg.sender, data);

            emit Deposit(_destinationChainId, _assetId, nonce);
        }
```

Security advice: None.

5.5. HydraBridge contract execute function【Pass】

Audit analysis: Perform security audit on the HydraBridge contract execute function. Its functional logic is: Proposal verification is performed according to the incoming _data. Only after the proposal status is confirm verification can it enter the fund pool for the execute cross-chain withdrawal function. After the audit, the authority control is correct and the logic design is reasonable.

```
    function execute(
        uint8 _originChainId,
        uint64 _lockNonce,
        bytes calldata _data,
```

```
        bytes32 _assetId
    ) external onlyObservers whenNotPaused {// knownsec // execute, called only by observers
        address vaultAddress = assetIdToVault[_assetId];
        bytes32 dataHash = keccak256(abi.encodePacked(vaultAddress, _data));
        uint72 nonceAndOriginChainId = (uint72(_lockNonce) << 8) | uint72(_originChainId);

        Proposal storage proposal = proposals[nonceAndOriginChainId][dataHash];
        require(proposal.status == ProposalStatus.Confirmed, "HydraBridge: Proposal not found or not confirmed");

        proposal.status = ProposalStatus.Completed;

        IVault vault = IVault(vaultAddress);
        vault.execute(_assetId, _data);

        emit StatusChanged(_originChainId, _lockNonce, ProposalStatus.Completed, dataHash);
    }
```

Security advice: None.

6. Code basic vulnerability detection

6.1. Compiler version security【Pass】

Check to see if a secure compiler version is used in the contract code implementation.

Detection results: After detection, the smart contract code has developed a compiler version of 0.8.6, there is no security issue.

Security advice: None.

6.2. Redundant code【Pass】

Check that the contract code implementation contains redundant code.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.3. Use of safe arithmetic library【Pass】

Check to see if the SafeMath security abacus library is used in the contract code implementation.

Detection results: The SafeMath security abacus library has been detected in the smart contract code and there is no such security issue.

Security advice: None.

6.4. Not recommended encoding【Pass】

Check the contract code implementation for officially uns recommended or deprecated coding methods.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.5. Reasonable use of require/assert【Pass】

Check the reasonableness of the use of require and assert statements in contract code implementations.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.6. Fallback function safety【Pass】

Check that the fallback function is used correctly in the contract code implementation.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.7. tx.origin authentication【Pass】

tx.origin is a global variable of 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 smart contracts makes contracts vulnerable to phishing-like attacks.z

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.8. Owner permission control【Pass】

Check that theowner in the contract code implementation has excessive permissions. For example, modify other account balances at will, and so on.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.9. Gas consumption detection【Pass】

Check that the consumption of gas exceeds the maximum block limit.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.10. call injection attack【Pass】

When a call function is called, strict permission control should be exercised, or the function called by call calls should be written directly to call calls.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.11. Low-level function safety【Pass】

Check the contract code implementation for security vulnerabilities in the use of call/delegatecall

The execution context of the call function is in the contract being called, while the execution context of the delegatecall function is in the contract in which the function is currently called.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.12. Vulnerability of additional token issuance【Pass】

Check to see if there are functions in the token contract that might increase the total token volume after the token total is initialized.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.13. Access control defect detection【Pass】

Different functions in the contract should set reasonable permissions, check whether the functions in the contract correctly use pubic, private and other keywords for visibility modification, check whether the contract is properly defined and use modifier access restrictions on key functions, to avoid problems caused by overstepping the authority.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.14. Numerical overflow detection【Pass】

The arithmetic problem in smart contracts is the integer overflow and integer overflow, with Solidity able to handle up to 256 digits (2^256-1), and a maximum number increase of 1 will overflow to get 0. Similarly, when the number is an unsigned type, 0 minus 1 overflows to get the maximum numeric value.

Integer overflows and underflows are not a new type of vulnerability, but they are particularly dangerous in smart contracts. Overflow conditions can lead to incorrect results, especially if the likelihood is not anticipated, which can affect the reliability and safety of the program.

Detection results: The security issue is not present in the smart contract code after

detection.

Security advice: None.

6.15. Arithmetic accuracy error【Pass】

Solidity has a data structure design similar to that of a normal programming language, such as variables, constants, arrays, functions, structures, and so on, and there is a big difference between Solidity and a normal programming language - Solidity does not have floating-point patterns, and all of Solidity's numerical operations result in integers, without the occurrence of decimals, and without allowing the definition of decimal type data. Numerical operations in contracts are essential, and numerical operations are designed to cause relative errors, such as sibling operations: 5/2 x 10 x 20, and 5 x 10/2 x 25, resulting in errors, which can be greater and more obvious when the data is larger.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.16. Incorrect use of random numbers【Pass】

Random numbers may be required in smart contracts, and while the functions and variables provided by Solidity can access significantly unpredictable values, such as block.number and block.timestamp, they are usually either more public than they seem, or are influenced by miners, i.e. these random numbers are somewhat predictable, so

malicious users can often copy it and rely on its unpredictability to attack the feature.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.17. Unsafe interface usage【Pass】

Check the contract code implementation for unsafe external interfaces, which can be controlled, which can cause the execution environment to be switched and control contract execution arbitrary code.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.18. Variable coverage【Pass】

Check the contract code implementation for security issues caused by variable overrides.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.19. Uninitialized storage pointer【Pass】

A special data structure is allowed in solidity as a strut structure, while local

variables within the function are stored by default using stage or memory.

The existence of store (memory) and memory (memory) is two different concepts, solidity allows pointers to point to an uninitialized reference, while uninitialized local stage causes variables to point to other stored variables, resulting in variable overrides, and even more serious consequences, and should avoid initializing the task variable in the function during development.

Detection results: After detection, the smart contract code does not have the problem.

Security advice: None.

6.20. Return value call verification【Pass】

This issue occurs mostly in smart contracts related to currency transfers, so it is also known as silent failed sending or unchecked sending.

In Solidity, there are transfer methods such as transfer(), send(), call.value(), which can be used to send tokens to an address, the difference being: transfer send failure will be throw, and state rollback; Call.value returns false when it fails to send, and passing all available gas calls (which can be restricted by incoming gas_value parameters) does not effectively prevent reentration attacks.

If the return values of the send and call.value transfer functions above are not checked in the code, the contract continues to execute the subsequent code, possibly with unexpected results due to token delivery failures.

Detection results: The security issue is not present in the smart contract code after

detection.

Security advice: None.

6.21. Transaction order dependency【Pass】

Because miners always get gas fees through code that represents an externally owned address (EOA), users can specify higher fees to trade faster. Since blockchain is public, everyone can see the contents of other people's pending transactions. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy its transactions at a higher cost to preempt the original solution.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.22. Timestamp dependency attack【Pass】

Block timestamps typically use miners' local time, which can fluctuate over a range of about 900 seconds, and when other nodes accept a new chunk, they only need to verify that the timestamp is later than the previous chunk and has a local time error of less than 900 seconds. A miner can profit from setting the timestamp of a block to meet as much of his condition as possible.

Check the contract code implementation for key timestamp-dependent features.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.23. Denial of service attack【Pass】

Smart contracts that are subject to this type of attack may never return to normal operation. There can be many reasons for smart contract denial of service, including malicious behavior as a transaction receiver, the exhaustion of gas caused by the artificial addition of the gas required for computing functionality, the misuse of access control to access the private component of smart contracts, the exploitation of confusion and negligence, and so on.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.24. Fake recharge vulnerability【Pass】

The transfer function of the token contract checks the balance of the transfer initiator (msg.sender) in the if way, when the balances < value enters the else logic part and return false, and ultimately does not throw an exception, we think that only if/else is a gentle way of judging in a sensitive function scenario such as transfer is a less rigorous way of coding.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.25. Reentry attack detection【Pass】

The call.value() function in Solidity consumes all the gas it receives when it is used to send tokens, and there is a risk of re-entry attacks when the call to the call tokens occurs before the balance of the sender's account is actually reduced.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.26. Replay attack detection【Pass】

If the requirements of delegate management are involved in the contract, attention should be paid to the non-reusability of validation to avoid replay attacks

In the asset management system, there are often cases of entrustment management, the principal will be the assets to the trustee management, the principal to pay a certain fee to the trustee. This business scenario is also common in smart contracts.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.27. Rearrangement attack detection【Pass】

A reflow attack is an attempt by a miner or other party to "compete" with a smart contract participant by inserting their information into a list or mapping, giving an attacker the opportunity to store their information in a contract.

Detection results: After detection, there are no related vulnerabilities in the smart contract code.

Security advice: None.

7. Appendix A: Security Assessment of Contract Fund Management

Contract fund management		
The type of asset in the contract	**The function is involved**	**Security risks**
Contract Token Assets	Lock、execute、deposit	None

Check the security of the management of **digital currency assets** transferred by users in the business logic of the contract. Observe whether there are security risks that may cause the loss of customer funds, such as **incorrect recording, incorrect transfer, and backdoor** withdrawal of the **digital currency assets** transferred into the contract.

KNOWNSEC
Blockchain Lab

Official Website

www.knownseclab.com

E-mail

blockchain@knownsec.com

WeChat Official Account