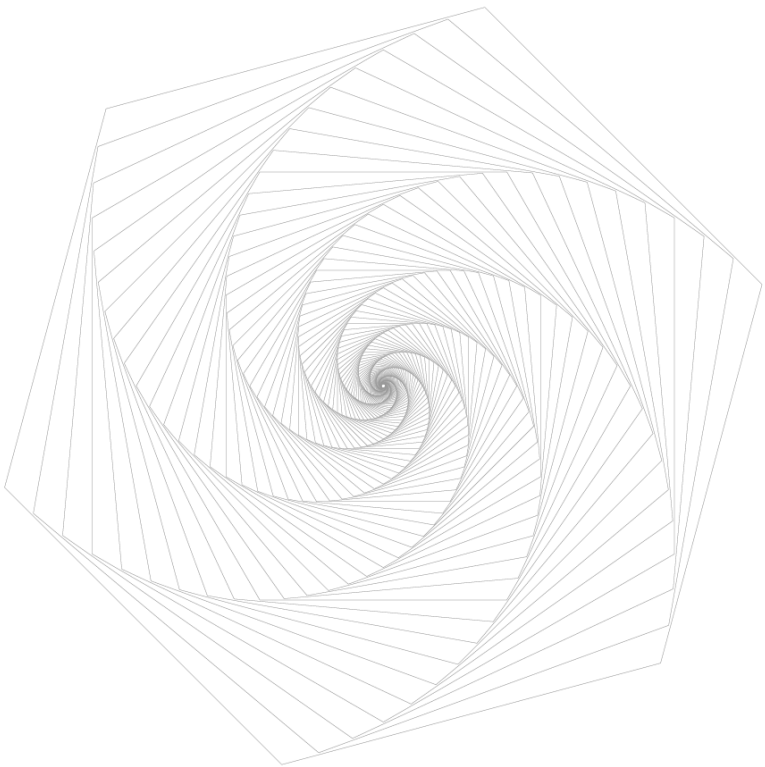
KNOWNSEC

Blockchain Lab

Smart Contract Audit Report



Version description

The revision	Date	Revised	Version
Write documentation	20220721	KNOWNSEC Blockchain Lab	V1.0

Document information

Title	Version	Document Number	Type
OrderNChaos Smart Contract Audit Report	V1.0		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 **July 19, 2022 to July 21, 2022**. During this period, the security and standardization of **the code of OrderNChaos smart contract code Bank, Chaos, GLA, Helper, Market, Order, prChaos, StakePool** will be audited and used as the statistical basis for the report.

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 OrderNChaos** 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	
report query link	

2. Item information

2.1. Item description

Briefly, OrderNChaos(ONC) is a twin system of algorithmic stable coins which borrows the idea of Nirvana on the Solana network, ONC protocol is implemented in solidity and can be run on any EVM compatible chains.

2.2. The project's website

https://ordernchaos.finance

2.3. White Paper

https://docs.ordernchaos.finance/overview/introduction

2.4. Review version code

2.5. Contract file and Hash/contract deployment address

The contract documents	MD5
Bank.sol	0FED1542314289D92BE1B32E35A2A18A
Chaos.sol	DDEBB8469C50F41DD52B57FB0B34E9B5
GLA.sol	920246D87A1804077EE5CB093FD2743B
Helper.sol	1E5B66C5F612FFB8B4ED8C8EF7B76B65
Market.sol	492CF42D40B82650EE03BA728F1D83E6
Order.sol	F741C237CFE35118B7A6BE67ABD71083

prChaos.sol	14C58B9D61841CE1486FE3D486A10D2B
StakePool.sol	21ADC7E5FB459EBE42660BCC20761C7B

3. External visibility analysis

3.1. Bank contracts

Bank					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
available	public	Flase	isInitialized	---	---
borrow	external	True	isInitialized	---	---
repay	external	True	isInitialized	---	---

3.2. Chaos contracts

Chaos					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
mint	public	True	onlyOwner	---	---
burn	public	True	onlyOwner	---	---
burnFrom	public	True	onlyOwner	---	---

3.3. GLA contracts

GLA					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
getPhase	external	Fasle	---	---	---
addWhitelist	external	True	onlyOwner	---	---
whitelistBuy	external	True	whitelistEnabled	---	---

3.4. **Helper contracts**

Helper					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
invest	public	True	override	---	---
reinvest	external	True	override	---	---
borrowAndInvest	public	True	override	---	---

3.5. **Market contracts**

Market					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
lowerAndAdjust	internal	True	---	---	---
setMarketOptions	external	True	override	---	---
addBuyStablecoin	external	False	override	---	---
manageStablecoins	external	True	onlyOwner	---	---
estimateBuy	public	False	isStarted	---	---
buyFor	public	True	override	---	---
realizeFor	public	True	onlyOwner	---	---
sellFor	public	True	onlyOwner	---	---
burnFor	public	True	onlyOwner	---	---

3.6. **Order contracts**

Order

funcName	visibility	state changes	decorator	payable reception	instructions
mint	public	True	onlyOwner	---	---
burnFrom	public	True	override	---	---

3.7. prChaos contracts

prChaos					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
mint	public	True	override	---	---
burn	public	True	override	---	---
burnFrom	public	True	override	---	---

3.8. StakePool contracts

StakePool					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
add	external	True	onlyOwner	---	---
pendingRewards	external	False	isInitialized	---	---
deposit	external	True	isInitialized	---	---
withdraw	external	True	isInitialized	---	---
claimFor	public	True	isInitialized	---	---
safeTransfer	internal	True	---	---	---

4. Code vulnerability analysis

4.1. Summary description of the audit results

Audit results			
audit project	**audit content**	**condition**	**description**
Business security detection	Bank.sol contract withdraws the rental token function	Pass	After testing, there is no security issue.
	GLA.sol contract whitelist related function	Pass	After testing, there is no security issue.
	GLA.sol contract request token function	Pass	After testing, there is no security issue.
	Helper.sol contract investment stablecoin function	Pass	After testing, there is no security issue.
	Market.sol contract stablecoin related function	Pass	After testing, there is no security issue.
	StakePool.sol contract pledge related 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	Reminder	After testing, tokens can be issued additionally.
	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. Bank.sol contract withdraws the rental token function 【Pass】

Audit analysis: The withdrawable function of the contract is used to calculate the amount of Chaos that users can withdraw. The available function and borrowFrom function of the contract are used to calculate the amount of Chaos that the current user can borrow. The function has the correct permissions and no obvious security issues have been found.

```
function withdrawable(address user, uint256 amountChaos) external view override isInitialized returns (uint256){
    uint256 userDebt = debt[user];
    uint256 floorPrice = market.f();
    if (amountChaos * floorPrice <= userDebt * 1e18) {//knownsec If the user's debt is greater than the token price, they will not borrow
        return 0;
    }
    return (amountChaos * floorPrice - userDebt * 1e18) / floorPrice;//knownsec Returns the number of tokens the user can withdraw
}

function available(address user) public view override isInitialized returns (uint256){
  uint256 userDebt = debt[user];
  (uint256 amountChaos, ) = pool.userInfo(0, user);
  uint256 floorPrice = market.f();
  if (amountChaos * floorPrice <= userDebt * 1e18) {//knownsec If the user's debt is greater than the token price, they will not borrow
    return 0;
  }
```

```
 return (amountChaos * floorPrice - userDebt * 1e18) / 1e18; //knownsec Returns the number of tokens the user can withdraw
}

function repay(uint256 amount) external override isInitialized whenNotPaused{
    require(amount > 0, "Bank: amount is zero");
    uint256 userDebt = debt[_msgSender()];//knownsec Get the user's current debt
    require(userDebt >= amount, "Bank: exceeds debt");
    Order.burnFrom(_msgSender(), amount);//knownsec Authorize Destruction Order
    unchecked {
        debt[_msgSender()] = userDebt - amount;
    }
    emit Repay(_msgSender(), amount);
}
```

Security advice: None.

5.2. GLA.sol contract whitelist related function【Pass】

Audit analysis: The addWhitelist function of the contract is used to add whitelist users, and the whitelistBuy function is used for whitelist users to purchase Chaos. The function has the correct permissions and no obvious security issues have been found.

```
function addWhitelist(address[] memory users) external beforeWhiteList onlyOwner{
    for (uint256 i = 0; i < users.length; i++) {
        address user = users[i];
        if (whitelistSharesOf[user] == 0) {//knownsec If not in the whitelist
            whitelistSharesOf[user] = EXISTS_IN_WHITELIST;
            whitelist.push(user);
        }
    }
}
```

```
function whitelistBuy(uint256 amount) external whitelistEnabled {
    require(amount > 0, "GLA: zero amount");
    uint256 shares = whitelistSharesOf[_msgSender()];
    require(shares != 0, "GLA: invalid whitelist user");//knownsec need to be in the whitelist
    shares = shares == EXISTS_IN_WHITELIST ? 0 : shares;
    uint256 maxCap = whitelistMaxCapPerUser - shares;
    if (amount > maxCap) {
        amount = maxCap;
    }
    require(amount > 0, "GLA: zero amount");
    USDC.safeTransferFrom(_msgSender(), address(this), amount);//knownsec Transfer USDC to the project party
    whitelistTotalShares += amount;
    whitelistSharesOf[_msgSender()] = shares + amount;
}
```

Security advice: None.

5.3. GLA.sol contract request token function【Pass】

Audit analysis: The claim function of the contract is used to claim Chaos. Although the Safetransfer function is not used, the maximum value of the transferred tokens is checked when the transfer function is called. Therefore, the function has the correct permissions and no obvious security problems have been found.

```
function claim() external isInitialized {
    uint256 chaos = estimateClaim(_msgSender());
    require(chaos > 0, "GLA: zero chaos");
    uint256 max = Chaos.balanceOf(address(this));
    Chaos.transfer(_msgSender(), max < chaos ? max : chaos);//knownsec Check if the maximum value of transferred tokens exceeds the threshold
```

```
    delete whitelistSharesOf[_msgSender()];
    delete publicOfferingSharesOf[_msgSender()];
}
```

Security advice: None.

5.4. Helper.sol contract investment stablecoin function【Pass】

Audit analysis: The contract's invest function and reinvest function are used for stablecoin investment, the borrowAndInvest function is used to borrow stablecoin Oreder to invest in volatility currency Chaos, and when buyFor function uses estimateBuy to buy volatility currency, a modifier is used to check whether the stablecoin can replace the volatility currency , the function permissions are correct, and no obvious security problems have been found.

```
function invest(address token,uint256 tokenWorth,uint256 desired,bool borrow) public override {
    IERC20(token).safeTransferFrom(_msgSender(), address(this), tokenWorth);
    IERC20(token).approve(address(market), tokenWorth);
    (uint256 chaos, ) = market.buyFor(token,tokenWorth,desired,_msgSender());//knownsec Check stablecoin origin by modifier canbuy
    Chaos.approve(address(pool), chaos);
    pool.depositFor(0, chaos, _msgSender());
    if (borrow) {
        borrowAndInvest((chaos * market.f()) / 1e18);
    }
}

function reinvest(address token,uint256 amount,uint256 desired) external override {
    prChaos.transferFrom(_msgSender(), address(this), amount);
    (, uint256 worth) = market.estimateRealize(amount, token); //knownsec Check stablecoin origin
```

```
by modifier canbuy
    IERC20(token).safeTransferFrom(_msgSender(), address(this), worth);
    IERC20(token).approve(address(market), worth);
    prChaos.approve(address(market), amount);
    market.realizeFor(amount, token, desired, _msgSender());
    Chaos.approve(address(pool), amount);
    pool.depositFor(0, amount, _msgSender());
}
```

Security advice: None.

5.5. Market.sol contract stablecoin related function【Pass】

Audit analysis: The canBuy modifier of the contract is used to check whether the specified stablecoin is eligible to exchange chaos, the addBuyStablecoin function is used to add stablecoins that can replace chaos, and the manageStablecoins function is used to manage stablecoins. The function has the correct permissions and no obvious security issues have been found.

```
modifier canBuy(address token) {
    require(stablecoinsCanBuy.contains(token), "Market: invalid buy token");
    _;
}

function addBuyStablecoin(address token) external onlyRole(ADD_STABLECOIN_ROLE){
    uint8 decimals = IERC20Metadata(token).decimals();
    require(decimals > 0, "Market: invalid token");
    stablecoinsDecimals[token] = decimals;
    stablecoinsCanBuy.add(token);//knownsec Add stablecoins
    emit StablecoinsChanged(token, true, true);
}
```

```
function manageStablecoins(address token, bool buyOrSell,bool addOrDelete) external override
onlyRole(MANAGER_ROLE) {
    if (addOrDelete) {
        uint8 decimals = IERC20Metadata(token).decimals();
        require(decimals > 0, "Market: invalid token");
        stablecoinsDecimals[token] = decimals;
        if (buyOrSell) {
            revert("Market: please call addBuyStablecoin!");
        } else {
            stablecoinsCanSell.add(token);
        }
        } else {
            if (buyOrSell) {
                stablecoinsCanBuy.remove(token);
            } else {
                stablecoinsCanSell.remove(token);
            }
        }
    emit StablecoinsChanged(token, buyOrSell, addOrDelete);
}
```

Security advice: None.

5.6. StakePool.sol contract pledge related function【Pass】

Audit analysis: The add function of the contract is used to add lp tokens, the pendingRewards function is used to view the rewards issued on the front end, the massUpdatePools and updatePool functions are used to update the pledge pool in batches, the depositFor and deposit are used to deposit tokens, and the withdraw

function is used to withdraw tokens. The function has the correct permissions and no obvious security issues have been found.

```
function add(uint256 _allocPoint,IERC20 _lpToken,bool _withUpdate) external override isInitialized onlyOwner {
    if (poolInfo.length == 0) {
        require(address(_lpToken) == address(Chaos),"StakePool: invalid lp token");
    }
    if (_withUpdate) {
        massUpdatePools();
    }
    totalAllocPoint = totalAllocPoint + _allocPoint;//knownsec Increase total distribution points
    poolInfo.push(PoolInfo({lpToken: _lpToken,allocPoint: _allocPoint,lastRewardBlock: block.number,accPerShare: 0}));
}

function massUpdatePools() public override isInitialized {
    uint256 totalSupply = Chaos.totalSupply();
    uint256 length = poolInfo.length;
    for (uint256 pid = 0; pid < length; ++pid) {//knownsec Batch update mining pool
        updatePool(pid, totalSupply);
    }
}

function depositFor(
            uint256 _pid,
            uint256 _amount,
            address _user
        ) public override isInitialized {
            PoolInfo storage pool = poolInfo[_pid];
            UserInfo storage user = userInfo[_pid][_user];
            updatePool(_pid, Chaos.totalSupply());
            if (user.amount > 0) {
```

```
            uint256 pending = (user.amount * pool.accPerShare) /
                1e12 -
                user.rewardDebt;
            if (pending > 0) {
                safeTransfer(_user, pending);
            }
        }
        pool.lpToken.safeTransferFrom(msg.sender, address(this), _amount);//knownsec
Transfer LP tokens
        user.amount = user.amount + _amount;
        user.rewardDebt = (user.amount * pool.accPerShare) / 1e12;//knownsec Calculate the
reward ratio
        emit Deposit(_user, _pid, _amount);
    }

    function withdraw(uint256 _pid, uint256 _amount)
            external
            override
            isInitialized
        {
            PoolInfo storage pool = poolInfo[_pid];
            UserInfo storage user = userInfo[_pid][msg.sender];
            require(user.amount >= _amount, "StakePool: withdraw not good");
            updatePool(_pid, Chaos.totalSupply());
            uint256 pending = (user.amount * pool.accPerShare) /
                1e12 -
                user.rewardDebt;
            if (pending > 0) {
                safeTransfer(msg.sender, pending);
            }
            uint256 fee = 0;
            if (_pid == 0) {
                uint256 withdrawable = bank.withdrawable(msg.sender,
```

```
user.amount);//knownsec Calculate the number of coins that can be withdrawn
                require(
                    withdrawable >= _amount,
                    "StakePool: amount exceeds withdrawable"
                );
                fee = (_amount * withdrawFee) / 10000;
            }

            user.amount = user.amount - _amount;
            user.rewardDebt = (user.amount * pool.accPerShare) / 1e12;
            pool.lpToken.safeTransfer(msg.sender, _amount - fee);
            pool.lpToken.safeTransfer(dev, fee);
            emit Withdraw(msg.sender, _pid, _amount - fee, fee);
        }
```

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 testing, the compiler version is greater than or equal to 0.8.0 in the smart contract code, and there is no such security problem.

```
1  // SPDX-License-Identifier: MIT
2  // OpenZeppelin Contracts v4.4.1 (utils/Address.sol)
3
4  pragma solidity ^0.8.0;
5
```

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 security issue is not present in the smart contract code after detection.

```
2705  library SafeMath {
2706      /**
2707       * @dev Returns the addition of two unsigned integers, with an overflow flag.
2708       *
2709       * _Available since v3.4._
2710       */
2711      function tryAdd(uint256 a, uint256 b) internal pure returns (bool, uint256) {
2712          unchecked {
2713              uint256 c = a + b;
2714              if (c < a) return (false, 0);
2715              return (true, c);
2716          }
2717      }
2718
2719      /**
2720       * @dev Returns the substraction of two unsigned integers, with an overflow flag.
2721       *
2722       * _Available since v3.4._
2723       */
2724      function trySub(uint256 a, uint256 b) internal pure returns (bool, uint256) {
2725          unchecked {
2726              if (b > a) return (false, 0);
2727              return (true, a - b);
2728          }
2729      }
```

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.

```
2987    function setTokenURIAffixes(string calldata _prefix, string calldata _suffix) external onlyOwner {
2988        tokenURIPrefix = _prefix;
2989        tokenURISuffix = _suffix;
2990    }
```

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【Reminder】

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: After testing, there is this problem in the smart contract code, Chaos and Order tokens can be issued by the owner. However, according to the OrderNChaos deployment specification, the owner of Order will be set to Bank, the owner of Chaos will be Market, and the MINT_ROLE of prChaos will be assigned to Bank. After that, the permissions of all contracts will be transferred to the Timelock contract, and OrderNChaos cannot issue additional tokens at any time. Need to wait for confirmation in Timelock. This part of the business logic is not in the contract, it belongs to the deployment script code and cannot be objectively determined, so a hint is given.

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.

```
2972    function _spawnAxie(uint256 _genes, address _owner) private returns (uint256 _axieId) {
2973        Axie memory _axie = Axie(_genes, block.timestamp);
2974        axies.push(_axie);
2975        _axieId = axies.length - 1;
2976        _safeMint(_owner, _axieId);
2977        emit AxieSpawned(_axieId, _owner, _genes);
2978    }
2979
```

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**
User Mortgage Token Assets	Invest, reinvest, borrowAndInvest	**PASS**
User mortgage platform currency assets	_mint, burn, burnFrom, add, deposit, withdraw	**PASS**

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