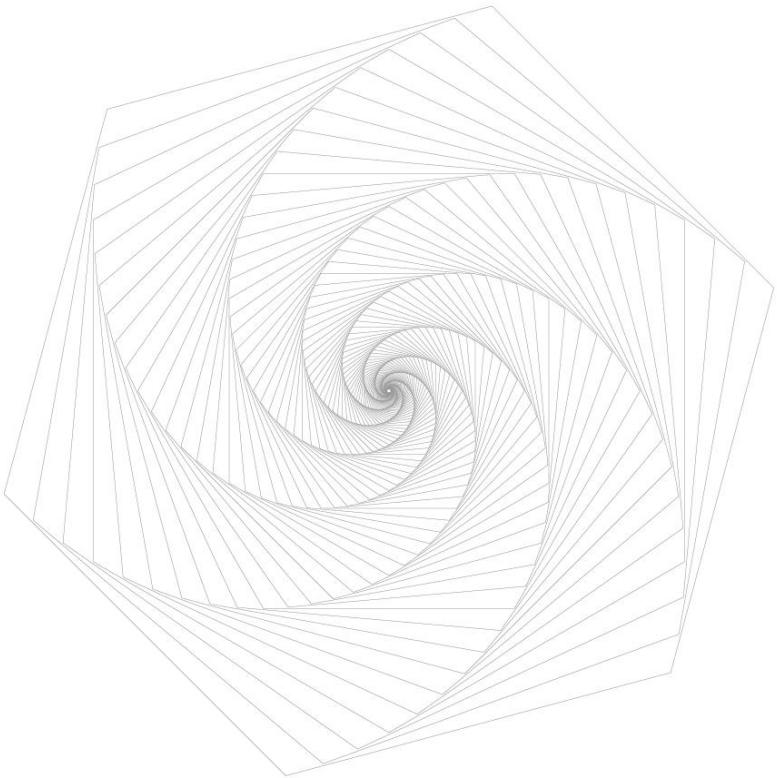
KNOWNSEC

Blockchain Lab

Smart contract audit report



Release Note

Modify Content	Time	Revised By	Version
Write Documentation	2024/2/28	Knownsec Blockchain Security Lab	V3.0

Document Information

Filename	Version	Report Number	Confidentia
JasperVault Smart Contract Audit Report	V3.0	bd37b7e5dd54	Team Public

Knownsec

Statement

Knownsec Blockchain Security Lab only issues this report based on facts that have occurred or existed before the issuance of this report, and assumes corresponding responsibilities. For facts that occur or exist after issuance, Knownsec Blockchain Security Lab cannot judge the security status of its smart contracts and is not responsible for them. The security description and other contents of this report are only based on the documents and information provided by the information provider to Knownsec Blockchain Security Lab as of the time of the issuance of this report. Knownsec Blockchain Security Lab assumes that the information provided is not missing, tampered with, deleted or concealed. If the provided information is missing, tampered with, deleted, concealed or the reflected situation is inconsistent with the actual situation, Knownsec Blockchain Security Lab will not be responsible for any losses and adverse effects caused thereby.

Knownsec

Table Of Content

1. Executive Summary ..... - 7 -

2. Project Information ..... - 8 -

2.1. Project Description ..... - 8 -

2.2. Audit Verion Code ..... - 9 -

2.3. Review Version Code ..... - 9 -

3. External Visibility Analysis ..... - 10 -

3.1. VaultFactory Contract ..... - 10 -

3.2. VaultPaymaster Contract ..... - 10 -

3.3. VaultManageModule Contract ..... - 11 -

3.4. IssuanceModule Contract ..... - 11 -

3.5. Manager Contract ..... - 12 -

3.6. OptionModule Contract ..... - 13 -

4. Code Vulnerability Analysis ..... - 14 -

4.1. Summary Of Audit Results ..... - 14 -

5. Detailed Description Of Findings ..... - 16 -

5.1. Fake underlying asset [Pass] ..... - 16 -

5.2. Fake underlying asset [Pass] ..... - 17 -

5.3. Fake receive asset [Pass] ..... - 18 -

5.4. Fake receive asset [Pass] ..... - 20 -

6. Basic Code Vulnerability Detection ..... - 22 -

Knownsec

6.1. Compiler version security [Pass] ..... - 22 -

6.2. Redundant code [Pass] ..... - 22 -

6.3. Use of secure arithmetic library [Pass] ..... - 22 -

6.4. Deprecated encoding method [Pass] ..... - 23 -

6.5. Reasonable use of require/assert [Pass] ..... - 23 -

6.6. Fallback function safety [Pass] ..... - 23 -

6.7. tx.origin authentication [Pass] ..... - 24 -

6.8. Owner permission control [Pass] ..... - 24 -

6.9. Gas consumption detection [Pass] ..... - 24 -

6.10. Call injection attack [Pass] ..... - 25 -

6.11. Low-level function safety [Pass] ..... - 25 -

6.12. Vulnerability of token issuance [Pass] ..... - 25 -

6.13. Access control defect detection [Pass] ..... - 26 -

6.14. Numerical overflow detection [Pass] ..... - 26 -

6.15. Arithmetic precision error [Pass] ..... - 27 -

6.16. Incorrect use of random numbers [Pass] ..... - 28 -

6.17. Unsafe external call [Pass] ..... - 28 -

6.18. Variable override [Pass] ..... - 29 -

6.19. Uninitialized storage pointer [Pass] ..... - 29 -

6.20. Return value call verification [Pass] ..... - 29 -

6.21. Transaction order dependence [Pass] ..... - 30 -

6.22. Timestamp dependency attack [Pass] ..... - 31 -

Knownsec

6.23. Denial of Service Attack [Pass] ..... - 32 -

6.24. Fake recharge vulnerability [Pass] ..... - 32 -

6.25. Reentrancy attack detection [Pass] ..... - 33 -

6.26. Replay attack detection [Pass] ..... - 33 -

6.27. Rearrangement attack detection [Pass] ..... - 34 -

7. Appendix: Contract Funds Management Security Assessment ... - 35 -

Knownsec

1. Executive Summary

This report covers the period from January 29, 2024 to February 28, 2024. During this period, the security and standardization of the JasperVault smart contract were audited and used as the statistical basis for the report.

This smart contract security audit excludes external contract calls, potential future attack methods, and code modifications resulting from contract upgrades or tampering. Notably, it also excludes considerations related to frontend security and server security.

During this test, Knownsec engineers conducted a comprehensive analysis of the common vulnerabilities of the smart contract (see Chapter 6) as well as the specific business security items related to the contract, and the comprehensive assessment was passed.

Since this test process is conducted in a non-production environment, all codes used are the latest backups. The test process is communicated with relevant interface personnel, and all relevant test operations are performed under controllable operational risks to avoid production errors during the testing process.

Knownsec

2. Project Information

2.1. Project Description

Jasper Vault, a peer-to-peer DeFi options protocol, leverages Account Abstraction (ERC-4337) technology to revolutionize options trading by eliminating counterparty risk and margin calls. By adopting this innovative approach, Jasper Vault transforms the DeFi landscape, offering a more secure and efficient trading experience for users.

The implementation of Account Abstraction (ERC-4337) in Jasper Vault allows users to directly lock their collateral in a smart contract vault, removing the need for a third-party custodian in options buying and writing. This fundamental shift mitigates counterparty risk, enhancing the security of the platform and empowering users to engage in options trading with confidence.

In Jasper Vault's peer-to-peer options model, the elimination of margin calls brings significant advantages. Without the need for margin calls, the protocol does not rely on external oracles to determine collateral values during the option tenor, simplifying the process and reducing potential vulnerabilities associated with external data sources.

Unlike traditional setups that require ongoing monitoring and potential liquidation of collateral, Jasper Vault collects interest upfront. This upfront collection guarantees writers receive their expected premium regardless of collateral value fluctuations during the option tenor, providing a stable and transparent revenue stream for writers and enhancing the overall protocol stability.

For buyers, Jasper Vault offers flexibility by allowing them to choose whether to exercise the option or default based on the underlying collateral's value at maturity. This flexibility empowers buyers to make strategic decisions aligned with market conditions and their financial situations, fostering a more dynamic and user-centric trading environment. Additionally, this approach aligns the incentives of buyers and writers,

Knownsec

giving buyers the opportunity to retain collateral if its value surpasses the initial amount, promoting a fair and mutually beneficial trading ecosystem within Jasper Vault.

2.2. Audit Verion Code

https://github.com/jaspervault/jasper-vaults-contracts.git#43da752

2.3. Review Version Code

https://github.com/jaspervault/jasper-vaults-contracts.git#bed0fc4

https://github.com/jaspervault/jasper-vaults-contracts#5493787

Knownsec

3. External Visibility Analysis

3.1. VaultFactory Contract

VaultFactory					
Function	Visibility	Status Modification	Modifier	Payable	Description
addStake	external	true	onlyOwner	payable	---
setVaultImplementation	public	true	onlyOwner	---	---
createAccount	public	true	---	---	---

3.2. VaultPaymaster Contract

VaultPaymaster					
Function	Visibility	Status Modification	Modifier	Payable	Description
setCostOfPost	external	true	onlyOwner	---	---
setQuotaWhiteList	external	true	---	---	---
deposit	external	true	---	---	---
depositEth	external	true	---	payable	---
withdrawEth	external	true	---	---	---
setOpenValidMiner	external	true	onlyOwner	---	---
setMinerList	external	true	onlyOwner	---	---
unlockStake	external	true	onlyOwner	---	---
withdrawStake	external	true	onlyOwner	---	---

Knownsec

addStake	external	true	onlyOwner	payable	---
postOp	external	true	---	---	---

3.3. VaultManageModule Contract

VaultManageModule					
Function	**Visibility**	**Status Modification**	**Modifier**	**Payable**	**Description**
removeVault	external	true	onlyVault	---	---
registToPlatform	external	true	---	---	---
setVaultMasterToken	external	true	onlyVault	---	---
setVaultProtocol	external	true	onlyVault	---	---
setVaultTokens	external	true	onlyVault	---	---
setVaultModule	external	true	onlyVault	---	---
setVaultType	external	true	onlyVault	---	---

3.4. IssuanceModule Contract

IssuanceModule					
Function	**Visibility**	**Status Modification**	**Modifier**	**Payable**	**Description**
setProxyIssueWhiteList	external	true	onlyVaultOrManager	---	---
issue	external	true	onlyVaultOrManager	---	---
issueAndProxy	external	true	---	payable	---

Knownsec

redeem	external	true	onlyVaultOrManager	---	---
redeemProxy	external	true	---	---	---

3.5. Manager Contract

Manager					
Function	**Visibility**	**Status Modification**	**Modifier**	**Payable**	**Description**
setPayer	external	true	onlyOwner	---	---
setWethAndEth	external	true	onlyOwner	---	---
setModules	external	true	onlyOwner	---	---
setProtocols	external	true	onlyOwner	---	---
setTokens	external	true	onlyOwner	---	---
setDBControlWhitelist	external	true	onlyOwner	---	---
transferOwnership	external	true	onlyOwner	---	---
diamondCut	external	true	onlyOwner	---	---
deleteFacetAllSelector	external	true	onlyOwner	---	---
setFuncWhiteList	external	true	onlyOwner	---	---
setFuncBlackList	external	true	onlyOwner	---	---
setCollateralNft	external	true	onlyOwner	---	---
setDomainHash	public	true	onlyOwner	---	---

Knownsec

3.6. OptionModule Contract

OptionModule					
Function	**Visibility**	**Status Modification**	**Modifier**	**Payable**	**Description**
submitPutOrder	external	true	---	---	---
liquidatePutOrder	external	true	---	payable	---
verifyCallOrder	external	true	---	---	---
submitCallOrder	external	true	---	---	---
liquidateCallOrder	external	true	---	payable	---

Knownsec

4. Code Vulnerability Analysis

4.1. Summary Of Audit Results

Audit Results			
Category	Assessment Item	Severity	Description
Business Security Audit	Fake underlying asset	Pass	After testing, this security issue has been confirmed and fixed.
	Fake receive asset	Pass	After testing, this security issue has been confirmed and fixed.
Basic Code Vulnerability Audit	Compiler version safety	Pass	After testing, there is no such security issue.
	Redundant code	Pass	After testing, there is no such security issue.
	Use of secure arithmetic libraries	Pass	After testing, there is no such security issue.
	Not recommended encoding method	Pass	After testing, there is no such security issue.
	Reasonable use of require/assert	Pass	After testing, there is no such security issue.
	Fallback function safety	Pass	After testing, there is no such security issue.
	tx.origin Authentication	Pass	After testing, there is no such security issue.
	Owner permission control	Pass	After testing, there is no such security issue.
	Gas consumption detection	Pass	After testing, there is no such security issue.
	Call injection attack	Pass	After testing, there is no such security issue.
	Low-level function	Pass	After testing, there is no such security issue.

	safety		
	Additional token issuance vulnerability	Hint	After testing, there is no such security issue.
	Access control flaw detection	Pass	After testing, there is no such security issue.
	Numeric overflow detection	Pass	After testing, there is no such security issue.
	arithmetic accuracy error	Pass	After testing, there is no such security issue.
	Error detection using random numbers	Pass	After testing, there is no such security issue.
	Unsafe external call	Pass	After testing, there is no such security issue.
	Variable override	Pass	After testing, there is no such security issue.
	Uninitialized storage pointer	Pass	After testing, there is no such security issue.
	Return value call verification	Pass	After testing, there is no such security issue.
	Transaction sequence dependency detection	Pass	After testing, there is no such security issue.
	Timestamp dependency attack	Pass	After testing, there is no such security issue.
	Denial of service attack detection	Pass	After testing, there is no such security issue.
	Fake recharge vulnerability detection	Pass	After testing, there is no such security issue.
	Reentrancy attack detection	Pass	After testing, there is no such security issue.
	Replay attack detection	Pass	After testing, there is no such security issue.
	Reorder attack detection	Pass	After testing, there is no such security issue.

5. Detailed Description Of Findings

5.1. Fake underlying asset [Pass]

Location: modules/OptionModule.sol

Severity: High

Description:

Due to the lack of detection methods for underlying asset, fake asset security means that attackers can use fake symbol tokens with the same name to enter the market, such as fake WBTC, WETH, etc., to confuse real tokens. This issue may lead to users purchasing fake assets and causing financial losses. The platform should conduct security address checks on all tokens entering the market to prevent such situations from occurring.

```
ftrace | funcSig
125     function submitPutOrder(
126         ILendFacet.PutOrder memory _putOrder,
127         bytes calldata _borrowerSignature,
128         bytes calldata _lenderSignature
129     ) external nonReentrant {
130         //verify data
131         verifyPutOrder(_putOrder);
```

PeckShield

```
64          //verify underlyingAsset
65          if (_putOrder⬆.underlyingAsset == eth) {
66              require(_putOrder⬆.optionHolder.balance >= _putOrder⬆.underlyingA
67          } else {
68              if (_putOrder⬆.underlyingAssetType == 0) {
69                  //verify token amount
70                  require(IERC20(_putOrder⬆.underlyingAsset).balanceOf(_putOrder
71              } else if (_putOrder⬆.underlyingAssetType == 1) {
72                  //verify uniswapv3 nft liquidity
73                  if (lendFacet.getCollateralNft(_putOrder⬆.underlyingAsset) ==
74                      (,,address token0,address token1,,,,uint128 liquidity,,,,)
75                      require(platformFacet.getTokenType(token0) != 0 &&  platfo
76                      require(uint256(liquidity) >= _putOrder⬆.underlyingAmount,
77                  } else {
78                      revert("OptionModule:invalid Nft");
79                  }
80              } else {
81                  revert("OptionModule:underlyingAssetType error");
82              }
83          }
```

Recommendations:

Suggest setting up a whitelist of underlying asset address to prevent fake asset from entering the market.

Fix Result:

Fixed at Github code commit#5493787

This security issue has been whitelisted for verification.

5.2. Fake underlying asset [Pass]

Location: modules/OptionModule.sol

Severity: High

Description:

Due to the lack of detection methods for underlying asset, fake asset security means that attackers can use fake symbol tokens with the same name to enter the market, such as fake WBTC, WETH, etc., to confuse

Knownsec

real tokens. This issue may lead to users purchasing fake assets and causing financial losses. The platform should conduct security address checks on all tokens entering the market to prevent such situations from occurring.

```
ftrace | funcSig
339     function submitCallOrder(
340         ILendFacet.CallOrder memory _callOrder⬆,
341         bytes calldata _borrowerSignature⬆,
342         bytes calldata _lenderSignature⬆
343     ) external nonReentrant {
344         verifyCallOrder(_callOrder⬆);
```

```
276     //verify underlyingAsset
277     if (_callOrder⬆.underlyingAsset == eth) {
278         require(_callOrder⬆.optionWriter.balance >= _callOrder⬆.underlyingAmount,"OptionModule:underlyingA
279     } else {
280         if (_callOrder⬆.underlyingAssetType == 0) {
281             require(IERC20(_callOrder⬆.underlyingAsset).balanceOf(_callOrder⬆.optionWriter) >= _callOrder⬆
282         } else if (_callOrder⬆.underlyingAssetType == 1) {
283             //verify uniswapv3 nft liquidity
284             if (lendFacet.getCollateralNft(_callOrder⬆.underlyingAsset) ==  ILendFacet.CollateralNftType.Un
285                 (,,address token0,address token1,,,,uint128 liquidity,,,,) = INonfungiblePositionManager(_ca
286                 require(platformFacet.getTokenType(token0) != 0 &&platformFacet.getTokenType(token1) != 0,"
287                 require( uint256(liquidity) >= _callOrder⬆.underlyingAmount,"OptionModule:underlyingAmount
288             } else {
289                 revert("OptionModule:invalid Nft");
290             }
291         } else {
292             revert("OptionModule:underlyingAssetType error");
293         }
294     }
```

Recommendations:

Suggest setting up a whitelist of underlying asset address to prevent fake asset from entering the market.

Fix Result:

Fixed at Github code commit#5493787

This security issue has been whitelisted for verification.

5.3. Fake receive asset [Pass]

Location: modules/OptionModule.sol

Severity: High

Description:

Due to the lack of detection methods for receive asset, fake asset security means that attackers can use fake symbol tokens with the same name to enter the market, such as fake WBTC, WETH, etc., to confuse real tokens. This issue may lead to users purchasing fake assets and causing financial losses. The platform should conduct security address checks on all tokens entering the market to prevent such situations from occurring.

```
ftrace | funcSig
125  function submitPutOrder(
126      ILendFacet.PutOrder memory _putOrder⬆,
127      bytes calldata _borrowerSignature⬆,
128      bytes calldata _lenderSignature⬆
129  ) external nonReentrant {
130      //verify data
131      verifyPutOrder(_putOrder⬆);
```

```
83          }
84          //verify receiveAsset
85          if (_putOrder⬆.receiveAsset == eth) {
86              require(_putOrder⬆.optionWriter.balance >= _putOrder⬆.receiveAmount,"Optio
87          } else {
88              require(IERC20(_putOrder⬆.receiveAsset).balanceOf(_putOrder⬆.optionWriter)
89          }
90      }
```

Recommendations:

Suggest setting up a whitelist of receive asset address to prevent fake asset from entering the market.

Fix Result:

Fixed at [Github code commit#5493787]

This security issue has been whitelisted for verification.

5.4. Fake receive asset [Pass]

Location: modules/OptionModule.sol

Severity: High

Description:

Due to the lack of detection methods for receive asset, fake asset security means that attackers can use fake symbol tokens with the same name to enter the market, such as fake WBTC, WETH, etc., to confuse real tokens. This issue may lead to users purchasing fake assets and causing financial losses. The platform should conduct security address checks on all tokens entering the market to prevent such situations from occurring.

```
ftrace | funcSig
339     function submitCallOrder(
340         ILendFacet.CallOrder memory _callOrder⬆,
341         bytes calldata _borrowerSignature⬆,
342         bytes calldata _lenderSignature⬆
343     ) external nonReentrant {
344         verifyCallOrder(_callOrder⬆);
```

```
295         //verify lendAsset
296         if (_callOrder⬆.optionPremiumAsset == eth) {
297             require( _callOrder⬆.optionHolder.balance >=(_callOrder⬆.optionPremiumAmount +
298         } else {
299             require( IERC20(_callOrder⬆.optionPremiumAsset).balanceOf(_callOrder⬆.optionHo
300         }
301     }
302
```

Recommendations:

Suggest setting up a whitelist of receive asset address to prevent fake asset from entering the market.

Fix Result:

Fixed at Github code commit#5493787

This security issue has been whitelisted for verification.

Knownsec

6. Basic Code Vulnerability Detection

6.1. Compiler version security [Pass]

Check whether a safe compiler version is used in the contract code implementation.

Status: After testing, the compiler version specified in the smart contract code is greater than 0.8.0, and there is no such security problem.

Recommendations: None.

6.2. Redundant code [Pass]

Check whether the contract code implementation contains redundant code.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.3. Use of secure arithmetic library [Pass]

Check whether the SafeMath arithmetic library is used in the contract code implementation.

Status: After testing, the SafeMath secure arithmetic library has been used in the smart contract code, and there is no such security issue.

Recommendations: None.

Knownsec

6.4. Deprecated encoding method [Pass]

Check whether there are officially deprecated or deprecated coding methods in the contract code implementation.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.5. Reasonable use of require/assert [Pass]

Check the rationality of the use of require and assert statements in the contract code implementation.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.6. Fallback function safety [Pass]

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

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

Knownsec

6.7. tx.origin authentication [Pass]

The tx.origin is a global variable in solidity that walks 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 makes the contract vulnerable to phishing-like attacks.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.8. Owner permission control [Pass]

Check whether the owner in the contract code implementation has excessive permissions. For example, arbitrarily modify other account balances, etc.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.9. Gas consumption detection [Pass]

Check whether the gas consumption exceeds the block maximum limit.

Status: After testing, there is no such security issue in the smart contract code.

Knownsec

Recommendations：None.

6.10. Call injection attack [Pass]

When the call function is called, strict permission control should be implemented, or the function called should be directly programmed.

Status：After testing, there is no such security issue in the smart contract code.

Recommendations：None.

6.11. Low-level function safety [Pass]

Check whether there are security vulnerabilities in the use of low-level functions (call/delegatecall) in the contract code implementation.

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 that is currently calling the function.

Status：After testing, there is no such security issue in the smart contract code.

Recommendations：None.

6.12. Vulnerability of token issuance [Pass]

Check whether there is a function in the token contract that may increase the total amount of tokens after initializing the total amount of

Knownsec

tokens.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.13. Access control defect detection [Pass]

Different functions in the contract should set reasonable permissions, check whether each function in the contract correctly uses keywords such as public and private for visibility modification, check whether the contract is correctly defined and use modifiers to restrict access to key functions, and avoid errors caused by overstepping of authority.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.14. Numerical overflow detection [Pass]

Arithmetic problems in smart contracts refer to integer overflow and integer underflow. Solidity can handle up to 256-bit numbers (2^256-1). Increasing the maximum number by 1 will overflow to 0. Likewise, when a number is an unsigned type, 0 minus 1 underflows to the maximum number value.

Integer overflows and underflows are not a new type of vulnerability,

Knownsec

but they are particularly dangerous in smart contracts. Overflow conditions can lead to incorrect results, especially if the possibility is not anticipated, and can affect the reliability and security of your program.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.15. Arithmetic precision error [Pass]

As a programming language, Solidity has a data structure design similar to that of ordinary programming languages, such as variables, constants, arrays, functions, structures, etc. There is also a big difference between Solidity and ordinary programming languages - Solidity does not have floating point types and the results of all numerical operations in Solidity will only be integers, and decimals will not occur. At the same time, decimal type data is not allowed to be defined. Numerical operations in contracts are essential, and the design of numerical operations may cause relative errors. For example, same-level operations: 5/2*10=20, and 5*10/2=25, thus causing errors. When the data is larger, the error will be larger and more obvious.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

Knownsec

6.16. Incorrect use of random numbers [Pass]

The use of random numbers may be required in smart contracts, and while Solidity provides functions and variables that can access apparently unpredictable values such as block.number and block.timestamp, they are often either more public than they appear, or are subject to miner influence, i.e. These random numbers are somewhat predictable, so a malicious user can often copy it and rely on its unpredictability to attack the feature.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.17. Unsafe external call [Pass]

Check whether unsafe external interfaces are used in contract code implementation. Controllable interfaces may cause the execution environment to be switched and control the contract to execute arbitrary code.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

Knownsec

6.18. Variable override [Pass]

Check whether there are security issues caused by variable coverage in the contract code implementation.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.19. Uninitialized storage pointer [Pass]

A special data structure is allowed in solidity, which is the struct structure, and local variables within functions are stored in storage or memory by default.

Storage and memory are two different concepts. Solidity allows a pointer to point to an uninitialized reference, and uninitialized local stroage will cause variables to point to other storage variables, leading to variable overwriting, or even more serious problems. As a result, you should avoid initializing struct variables in functions during development.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.20. Return value call verification [Pass]

This issue often occurs in smart contracts related to currency

Knownsec

transfers, so it is also called silent failed sending or unchecked sending.

In solidity, there are transfer(), send(), call.value() and other currency transfer methods, all of which can be used to send tokens to a certain address. The difference is: when the transfer fails, a throw will be thrown and the status will be rolled back. Only 2300 gas will be passed for calling to prevent reentrancy attacks, false will be returned when send fails, only 2300 gas will be passed for calling to prevent reentrancy attacks; false will be returned when call.value fails to be sent; all available gas will be passed for calling (It can be restricted by passing in the gas_value parameter), but it cannot effectively prevent reentrancy attacks.

If the return value of the above send and call.value transfer functions is not checked in the code, the contract will continue to execute the subsequent code, which may lead to unexpected results due to failure to send tokens.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.21. Transaction order dependence [Pass]

Since miners always receive gas fees through code representing an externally owned address (EOA), users can specify higher fees to conduct transactions faster. Since the blockchain is public, everyone can see the

Knownsec

contents of every pending transaction. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy their transactions at a higher fee to preempt the original solution.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.22. Timestamp dependency attack [Pass]

The timestamp of the data block usually uses the local time of the miner, and this time can fluctuate within a range of approximately 900 seconds. When other nodes accept a new block, they only need to verify whether the timestamp is later than the previous block and The error is within 900 seconds from the local time. A miner can profit from this by setting the timestamp of a block to satisfy the conditions in his favor as much as possible.

Check whether there are key functions that depend on timestamps in the contract code implementation.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

Knownsec

6.23. Denial of Service Attack [Pass]

A smart contract that suffers this type of attack may never return to normal working order. There may be many reasons for a denial of service in a smart contract, including malicious behavior when serving as the recipient of a transaction, artificially increasing the gas required for computing functions leading to gas exhaustion, abusing access control to access private components of the smart contract, exploiting confusion and negligence, etc.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.24. Fake recharge vulnerability [Pass]

The transfer function of the token contract uses the if judgment method to check the balance of the transfer initiator (msg.sender). When balances[msg.sender] < value, it enters the else logic part and returns false. In the end, no exception was thrown. We believe that the mild judgment method of if/else is an imprecise coding method in sensitive function scenarios such as transfer.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

Knownsec

6.25. Reentrancy attack detection [Pass]

The call.value() function in solidity consumes all the gas it receives when it is used to send tokens. When the call.value() function is called, the operation of sending tokens occurs before the balance of the sender's account is actually reduced, there is a risk of reentrancy attacks.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

6.26. Replay attack detection [Pass]

If the contract involves the need for entrusted management, attention should be paid to the non-reusability of verification to avoid replay attacks.

In the asset management system, there are often cases of entrusted management. The client gives the assets to the trustee for management, and the client pays a certain fee to the trustee. This business scenario is also relatively common in smart contracts.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

Knownsec

6.27. Rearrangement attack detection [Pass]

A reordering attack occurs when miners or other parties attempt to "compete" with smart contract participants by inserting their own information into a list or mapping, thereby giving the attacker the opportunity to store their own information into the contract middle.

Status: After testing, there is no such security issue in the smart contract code.

Recommendations: None.

Knownsec

7. Appendix: Contract Funds Management Security Assessment

Contract Funds Management		
Asset Type In Contract	**Involving Function**	**Security Risk**
User staked assets management	VaultPaymaster.deposit, VaultPaymaster.depositEth, VaultPaymaster.withdrawEth	Tip: It depends on the specific functions of the business
Assets management	---	Tip: It depends on the specific functions of the business

The management security of digital currency assets transferred by users in the contract business logic. Observe whether there are security risks such as incorrect recording of digital currency assets transferred into the contract, incorrect transfer out, backdoor withdrawals, etc., that may easily cause the loss of customer funds.

Knownsec

KNOWNSEC

Blockchain Lab

Official Website

www.knownseclab.com

E-mail

blockchain@knownsec.com

WeChat Official Account