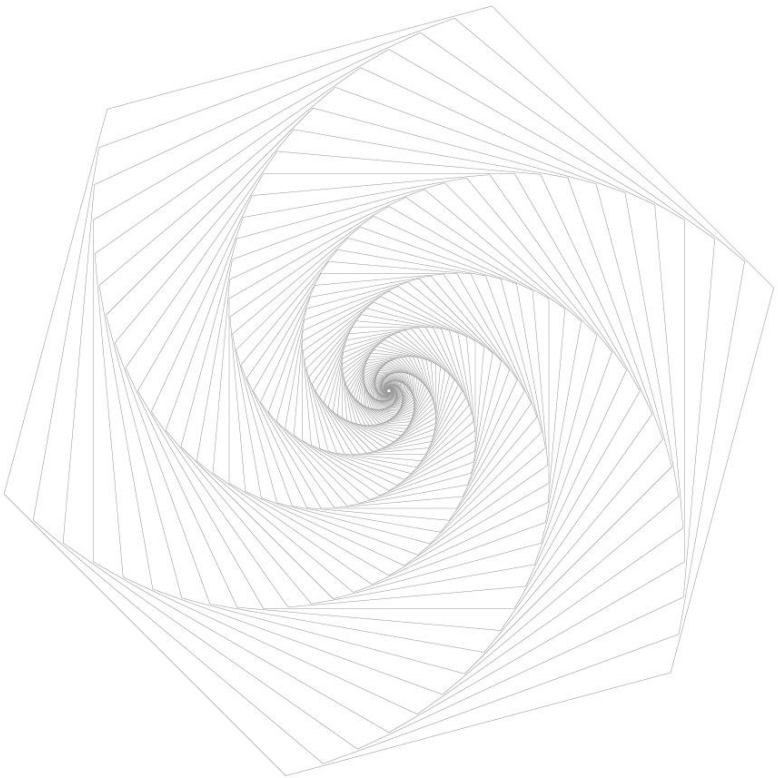
KNOWNSEC

Blockchain Lab

Smart contract audit report



Release Note

Modify Content	Time	Revised By	Version
Write Documentation	2024/6/28	Knownsec Blockchain Security Lab	V4.1

Document Information

Filename	Version	Report Number	Confidentiality
JasperVault Smart Contract Audit Report	V4.1	bd37b7e5dd54	Team Public

Knownsec

Statement

Knownsec Blockchain Security Lab issues this report solely based on facts that have occurred or existed prior to its issuance and assumes corresponding responsibilities for these facts. The Lab cannot assess the security status of its smart contracts for events occurring or existing after the report's issuance and is not liable for such matters. The security description and other content of this report are based solely on the documents and information provided by the information provider to Knownsec Blockchain Security Lab up to the report's issuance date. The Lab assumes that the provided information is complete, unaltered, and accurately reflects the actual situation. If the information is found to be missing, tampered with, deleted, or concealed, or if it does not accurately represent the actual situation, Knownsec Blockchain Security Lab will not be held responsible for any losses or adverse effects arising from these discrepancies.

Knownsec

Table Of Content

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

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

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

2.2. Audit Version 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. Manager Contract ..... - 11 -

3.4. PriceOracle Contract ..... - 12 -

3.5. Vault Contract ..... - 12 -

3.6. VaultManageModule Contract ..... - 13 -

3.7. IssuanceModule Contract ..... - 13 -

3.8. OptionModule Contract ..... - 14 -

3.9. TradeModule Contract ..... - 14 -

3.10. PriorityPool Contract ..... - 15 -

3.11. RegularPool Contract ..... - 15 -

3.12. OptionService Contract ..... - 15 -

4. Code Vulnerability Analysis ..... - 17 -

4.1. Summary of Audit Results ..... - 17 -

Knownsec

5. Detailed Description of Findings ..... - 20 -

5.1. Lack of Underlying Asset Verification [Pass] ..... - 20 -

5.2. Lack of Underlying Asset Verification [Pass] ..... - 21 -

5.3. Lack of Receive Asset Verification [Pass] ..... - 22 -

5.4. Lack of Receive Asset Verification [Pass] ..... - 24 -

5.5. Lack of Oracle Price Verification [Pass] ..... - 25 -

5.6. Reentrancy Attack Risk [Pass] ..... - 26 -

6. Basic Code Vulnerability Detection ..... - 28 -

6.1. Compiler Version Security [Pass] ..... - 28 -

6.2. Redundant Code [Pass] ..... - 28 -

6.3. Use of Secure Arithmetic Library [Pass] ..... - 28 -

6.4. Deprecated Encoding Method [Pass] ..... - 29 -

6.5. Reasonable Use of Require/Assert [Pass] ..... - 29 -

6.6. Fallback Function Safety [Pass] ..... - 29 -

6.7. Authentication of Tx.origin [Pass] ..... - 30 -

6.8. Owner Permission Control [Pass] ..... - 30 -

6.9. Gas Consumption Detection [Pass] ..... - 30 -

6.10. Call Injection Attack [Pass] ..... - 31 -

6.11. Low-level Function Safety [Pass] ..... - 31 -

6.12. Vulnerability of Token Issuance [Pass] ..... - 31 -

6.13. Access Control Defect Detection [Pass] ..... - 32 -

6.14. Numerical Overflow Detection [Pass] ..... - 32 -

Knownsec

6.15. Arithmetic Precision Error [Pass] ..... - 33 -

6.16. Incorrect Use of Random Numbers [Pass] ..... - 34 -

6.17. Unsafe External Call [Pass] ..... - 34 -

6.18. Variable Override [Pass] ..... - 35 -

6.19. Uninitialized Storage Pointer [Pass] ..... - 35 -

6.20. Return Value Call Verification [Pass] ..... - 35 -

6.21. Transaction Order Dependence [Pass] ..... - 36 -

6.22. Timestamp Dependency Attack [Pass] ..... - 37 -

6.23. Denial of Service Attack [Pass] ..... - 38 -

6.24. Fake Recharge Vulnerability [Pass] ..... - 38 -

6.25. Reentrancy Attack Detection [Pass] ..... - 39 -

6.26. Replay Attack Detection [Pass] ..... - 39 -

6.27. Rearrangement Attack Detection [Pass] ..... - 40 -

7. Appendix: Security Analysis for Contract Fund Management ..... - 41 -

Knownsec

1. Executive Summary

This report covers the period from June 17, 2024, to June 28, 2024. During this period, the security and standardization of the JasperVault smart contract were audited, and the results of the audit served as the statistical basis for the report.

This smart contract security audit excludes considerations of external contract calls, potential future attack vectors, and code modifications that may arise from contract upgrades or tampering. Notably, it also does not account for issues related to frontend or 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, it utilizes the most recent code backups. The process is communicated to relevant interface personnel, and all relevant test operations are conducted under controlled operational risks to prevent errors from affecting the production environment.

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 Version Code

[V1.0]https://github.com/jaspervault/jasper-vaults-contracts/commit/43da752d3c94958e2e491b2c3b52bdae1ed0e046

2.3. Review Version Code

[V2.0]https://github.com/jaspervault/jasper-vaults-contracts/commit/bed0fc42a8cf0e87a621c66ff82ed47c191e6517

[V3.0]https://github.com/jaspervault/jasper-vaults-contracts/commit/5493787e9ba600f032bd11516e68534eeddb0d46

[V4.0]https://github.com/jaspervault/jasper-vaults-contracts/commit/97e04642c2a0e87a82e19fde5aa99635aa128b37

Knownsec

3. External Visibility Analysis

3.1. VaultFactory Contract

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

3.2. VaultPaymaster Contract

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

Knownsec

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

3.3. Manager Contract

Manager					
Function	**Visibility**	**State 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	---	---

Knownsec

3.4. PriceOracle Contract

PriceOracle					
Function	**Visibility**	**State Modification**	**Modifier**	**Payable**	**Description**
setPrice	external	true	IsVault	---	---
setOracles	external	true	onlyOwner	---	---

3.5. Vault Contract

Vault					
Function	**Visibility**	**State Modification**	**Modifier**	**Payable**	**Description**
execute	external	true	---	---	---
executeBatch	external	true	---	---	---
addDeposit	public	true	---	payable	---
withdrawDepositTo	public	true	onlyOwner	---	---
upgradeTo	public	true	onlyProxy	---	---
upgradeToAndCall	public	true	onlyProxy	payable	---

Knowsec

3.6. VaultManageModule Contract

VaultManageModule					
Function	**Visibility**	**State Modification**	**Modifier**	**Payable**	**Description**
removeVault	external	true	onlyVault	---	---
validVaultModuleV2	external	true	---	---	---
registToPlatform	external	true	vault.owner	---	---
setVaultMasterToken	external	true	onlyVault	---	---
setVaultProtocol	external	true	onlyVault	---	---
setVaultTokens	external	true	onlyVault	---	---
setVaultModule	external	true	onlyVault	---	---
setVaultType	external	true	onlyVault	---	---

3.7. IssuanceModule Contract

IssuanceModule					
Function	**Visibility**	**State Modification**	**Modifier**	**Payable**	**Description**
setProxyIssueWhiteList	external	true	onlyVaultOrManager	---	---
issue	external	true	onlyVaultOrManager	payable	---
issueAndProxy	external	true	getProxyIssueWhiteList	payable	---
redeem	external	true	onlyVaultOrManager	---	---
redeemProxy	external	true	---	---	---

Knownsec

3.8. OptionModule Contract

OptionModule					
Function	**Visibility**	**State Modification**	**Modifier**	**Payable**	**Description**
setOptionService	external	true	onlyOwner	---	---
setOracleWhiteList	external	true	onlyOwner	---	---
submitJvaultOrderSingle	external	true	onlyVaultOrManager	---	---
submitOptionOrder	external	true	onlyVaultOrManager	---	---
submitHostingOptionOrder	external	true	onlyVaultOrManager	---	---

3.9. TradeModule Contract

TradeModule					
Function	**Visibility**	**State Modification**	**Modifier**	**Payable**	**Description**
trade	external	true	onlyVault	---	---

Knownsec

3.10. PriorityPool Contract

PriorityPool					
Function	**Visibility**	**State Modification**	**Modifier**	**Payable**	**Description**
deposit	external	true	nonReentrant	payable	---
withdrawProfit	external	true	nonReentrant	---	---
withdrawPrincipal	external	true	nonReentrant	---	---

3.11. RegularPool Contract

RegularPool					
Function	**Visibility**	**State Modification**	**Modifier**	**Payable**	**Description**
deposit	external	true	nonReentrant	payable	---
withdraw	external	true	nonReentrant	---	---

3.12. OptionService Contract

OptionService					
Function	**Visibility**	**State Modification**	**Modifier**	**Payable**	**Description**
setPriceOracle	external	true	onlyOwner	---	---
createPutOrder	external	true	onlyModule	---	---

Knownsec

createCallOrder	external	true	onlyModule	---	---
liquidateOption	external	true	nonReentrant	payable	---

Knownsec

4. Code Vulnerability Analysis

4.1. Summary of Audit Results

Audit Results			
Category	Assessment Item	Severity Level	Description
Business Security Audit	Lack of Underlying Asset Verification	Pass	After testing, this security issue has been confirmed and fixed.
	Lack of Receive Asset Verification	Pass	After testing, this security issue has been confirmed and fixed.
	Lack of Oracle Price Verification	Pass	After testing, this security issue has been confirmed and fixed.
	Reentrancy Attack Risk	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	Pass	After testing, there is no such security issue.

	control		
	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 safety	Pass	After testing, there is no such security issue.
	Additional token issuance vulnerability	Pass	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.

Knownsec

5. Detailed Description of Findings

5.1. Lack of Underlying Asset Verification [Pass]

Severity Level: High

Code Location: modules/OptionModule.sol

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);
```

PROWNSEC

```
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. Lack of Underlying Asset Verification [Pass]

Severity Level: High

Code Location: modules/OptionModule.sol

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. Lack of Receive Asset Verification [Pass]

Severity Level: High

Code Location: modules/OptionModule.sol

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
function submitPutOrder(
    ILendFacet.PutOrder memory _putOrder⬆,
    bytes calldata _borrowerSignature⬆,
    bytes calldata _lenderSignature⬆
) external nonReentrant {
    //verify data
    verifyPutOrder(_putOrder⬆);
```

```
    }
    //verify receiveAsset
    if (_putOrder⬆.receiveAsset == eth) {
        require(_putOrder⬆.optionWriter.balance >= _putOrder⬆.receiveAmount,"Optio
    } else {
        require(IERC20(_putOrder⬆.receiveAsset).balanceOf(_putOrder⬆.optionWriter)
    }
}
```

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. Lack of Receive Asset Verification [Pass]

Severity Level: High

Code Location: modules/OptionModule.sol

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.

5.5. Lack of Oracle Price Verification [Pass]

Severity Level: Low

Code Location: adapters/oracle/ChainLinkOracleAdapter.sol

Description:

The function does not account for the validation of token prices returned by the oracle in certain special circumstances. In unexpected situations where the oracle does not return data, the token price tends toward zero; however, under normal conditions, the token price is expected to have a non-zero value.

```
ftrace | funcSig
66      function read(
67          address _masterToken⬆,
68          address _quoteToken⬆
69      ) external view returns (uint256) {
70          uint256 price;
71          if (_quoteToken⬆ == usdToken) {
72              return getPriceByBase(_masterToken⬆, usdToken);
73          }
74          if (_quoteToken⬆ == ethToken) {
75              return getPriceByBase(_masterToken⬆, ethToken);
76          }
77          (, price) = getPriceByUsd(_masterToken⬆, _quoteToken⬆);
78          return price;
```

Recommendations:

It is recommended to perform a security check on the price returned by Oracle. Generally, a token price should be greater than zero.

Fix Result:

Fixed at Github code commit#7339c06

This security issue has been fixed.

5.6. Reentrancy Attack Risk [Pass]

Severity Level: Low

Code Location:

modules/OptionModule.sol#L133, L258, L473

modules/PriorityPool.sol#L49, L95, L113

modules/RegularPool.sol#L44, L76

Description:

Reentrancy attacks are a type of security vulnerability in smart contracts where an attacker exploits a contract's interaction with external calls, allowing the attacker to repeatedly call a function before its initial execution is completed, potentially leading to unauthorized access, data corruption, or loss of funds.

```
46      * depoist to vault
47      * @param _amount deposit asset amount include:ETH,USDT,WETH
48      */
        ftrace | funcSig
49      function deposit(
50          uint256 _amount
51      ) payable external {
52
53          // deposit to vault
54          depositToVault(_amount);
55
```

Recommendations:

Implement rigorous reentrancy guards and checks in smart contracts to

Knownsec

prevent unauthorized re-entry and ensure that external calls are handled securely, thereby mitigating the risk of reentrancy attacks..

Fix Result:

Fixed at Github code commit#61cb36a

This security issue has been fixed.

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. Authentication of Tx.origin [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 an uninitialized local storage 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 of 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: Security Analysis for Contract Fund Management

Contract Fund Management		
Asset Type In Contract	**Involving Function**	**Security Risk**
User Staked Asset Management	VaultPaymaster.deposit, VaultPaymaster.depositEth, VaultPaymaster.withdrawEth PriorityPool.deposit PriorityPool.withdrawPrincipal RegularPool.deposit RegularPool.withdraw	Tip: It depends on the specific functions of the business
Asset 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