SMART CONTRACT AUDIT REPORT

for

Kalmar Bond

Prepared By: Patrick Lou

PeckShield

April 29, 2022

Document Properties

Client	Kalmar Protocol
Title	Smart Contract Audit Report
Target	Kalmar Bond
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 29, 2022	Xiaotao Wu	Final Release
1.0-rc	April 1, 2022	Xiaotao Wu	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction **4**

1.1 About Kalmar Bond 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 7

2 Findings **9**

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results **11**

3.1 Possible Price manipulation For _kalmPrice()/_getLpPrice() 11

3.2 Trust Issue of Admin Keys 12

4 Conclusion **15**

References **16**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Kalmar Bond feature in the Kalmar protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About Kalmar Bond

Kalmar is a decentralized bank powered by DeFi and NFT. The protocol uses secure financial instruments and advanced gamification models to make banking engaging, transparent and accessible. The audited Kalmar Bond is a treasury management strategy. The intention of bonds mechanics is to deepen the protocol's liquidity while incentivizing arbitrage opportunities. A limited daily supply of discounted platform's native tokens is available for a purchase with KALM LP tokens.

The basic information of the audited feature is as follows:

Table 1.1: Basic Information of Kalmar Bond

Item	Description
Name	Kalmar Protocol
Website	https://kalmar.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 29, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/kalmar-io/kalmar-bonding-contracts.git (bb6665d)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ **Likelihood**	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Kalmar Bond` feature. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	0	
Informational	0	
Total	2	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability and 1 medium-severity vulnerability.

Table 2.1: Key Kalmar Bond Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Possible Price manipulation For _kalmPrice()/_getLpPrice()	Time and State	Resolved
PVE-002	Medium	Trust Issue of Admin Keys	Security Features	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Possible Price manipulation For _kalmPrice()/_getLpPrice()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `KalmarBondingStrategy`
- Category: Time and State [4]
- CWE subcategory: CWE-362 [2]

Description

The `KalmarBondingStrategy` contract defines two functions (i.e., `_kalmPrice()` and `_getLpPrice()`) to obtain the prices of `kalm Token` and `lp Token`. During the analysis of these two functions, we notice the prices of `kalm Token/lp Token` are possible to be manipulated. In the following, we use the `_kalmPrice()` routine as an example.

To elaborate, we show below the related code snippet of the `KalmarBondingStrategy` contract. Specifically, if we examine the implementation of the `_kalmPrice()`, the final price of the `kalm Token` is derived from `(otherPERkalm*_usdTokenPrice())/(10**decimalsOther)` (line 236), where the value of `otherPERkalm` is calculated by `(1e18*otherReserve)/kalmReserve`. Although the price of `BUSD` is obtained from the `chainlink` and cannot be manipulated, `kalmReserve` or `otherReserve` is the token amount in the pool thus can be manipulated by flash loans, which causes the final values of the `kalm Token` not trustworthy.

```
228     function _kalmPrice() internal view returns (uint256) {
229         IPancakeswapV2Pair pair = IPancakeswapV2Pair(lp);
230         address other = pair.token0() == kalm ? pair.token1() : pair.token0();
231         (uint256 Res0, uint256 Res1, ) = pair.getReserves();
232         (uint256 kalmReserve, uint256 otherReserve) = pair.token0() == kalm ? (Res0,
                Res1) : (Res1, Res0);
233         uint256 decimalsOther = IERC20Detailed(other).decimals();
234         // amount
235         uint256 otherPERkalm = (1e18*otherReserve)/kalmReserve;
```

```
        uint256 kalmPrice = (otherPERkalm*_usdTokenPrice())/(10**decimalsOther);

        return kalmPrice;
    }
```

Listing 3.1: `KalmarBondingStrategy::_kalmPrice()`

Recommendation Revise current execution logic of `_kalmPrice()/_getLpPrice()` to defensively detect any manipulation attempts in the `kalm Token/lp Token` prices.

Status This issue has been fixed in this commit: `bcccff8`.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `KalmarBondingStrategy`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `Kalmar Bond` feature, there is a privileged account, i.e., `owner`. This account plays a critical role in governing and regulating the system-wide operations (e.g., update the `bondingEmission`, update the `kalm` price for the `KalmOraclePrice` contract, pause/unpause the bond buying, set the `burnAddress`, and rescue `ERC20` tokens from the `KalmarBondingStrategy` contract, etc.). Our analysis shows that this privileged account need to be scrutinized. In the following, we show the representative functions potentially affected by the privileges of the `owner` account.

```
    function updateBondingEmission(
      uint256 _startBondingTime,
      uint256 _endBondingTime,
      uint256 _maxBondingSell,
      uint256 _discount
    ) external onlyOwner {
        uint256 length = bondingEmission.length;
        require(_maxBondingSell < BONDPERDAY_MAX && _maxBondingSell >= BONDPERDAY_MIN, "Bond sell in limit amount.");
        /* require(block.timestamp > bondingEmission[length-1].endBondingTime, "Not finished last bonding time yet."); */
        require(_endBondingTime > _startBondingTime, "endTime > startTime!");
        bondingEmission.push(
          BondingEmission({
                    index: length,
                    startBondingTime: _startBondingTime,
                    endBondingTime: _endBondingTime,
```

```
                maxBondingSell: _maxBondingSell,
                currentSold: 0,
                discount: _discount
            })
        );
        emit UpdatedBondingEmission(length,_startBondingTime,_endBondingTime,
            _maxBondingSell);
    }
```

Listing 3.2: `KalmarBondingStrategy::updateBondingEmission()`

```
    /**
     * @notice Sets burn address
     * @dev Only callable by the contract admin.
     */
    function setBurnAddress(address _burnAddr) external onlyOwner {
        burnAddress = _burnAddr;
        emit BurnAddressSet(_burnAddr);
    }

    /**
     * @notice Triggers stopped state
     * @dev Only possible when contract not paused.
     */
    function pause() external onlyOwner whenNotPaused {
        _pause();
        emit Pause();
    }

    /**
     * @notice Returns to normal state
     * @dev Only possible when contract is paused.
     */
    function unpause() external onlyOwner whenPaused {
        _unpause();
        emit Unpause();
    }
```

Listing 3.3: `KalmarBondingStrategy::setBurnAddress()/pause()/unpause()`

```
    // Added to support recovering LP Rewards from other systems such as BAL to be
        distributed to holders
    function recoverERC20(address tokenAddress, uint256 tokenAmount) external onlyOwner
        {
        require(tokenAddress != address(stakingToken), "Cannot withdraw staking token");
        IERC20(tokenAddress).safeTransfer(owner(), tokenAmount);
        emit Recovered(tokenAddress, tokenAmount);
    }
```

Listing 3.4: `KalmarBondingStrategy::recoverERC20()`

Note that the `recoverERC20()` routine allows for the `owner` to withdraw any `ERC20` token from the `KalmarBondingStrategy` contract except for the `stakingToken`. If all the `kalm` tokens are withdrawn by the owner, the bond buyers will be unable to claim their kalm tokens from the KalmarBondingStrategy contract.

If the privileged owner account is a plain EOA account, this may be worrisome and pose counter-party risk to the protocol users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation. Moreover, it should be noted if current contracts are to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been fixed in this commit: 0674801.

4 Conclusion

In this audit, we have analyzed the `Kalmar Bond` design and implementation. `Kalmar Bond` is a treasury management strategy. The intention of bonds mechanics is to deepen the protocol's liquidity while incentivizing arbitrage opportunities. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[4] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

[5] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[6] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[7] PeckShield. PeckShield Inc. https://www.peckshield.com.