SMART CONTRACT AUDIT REPORT

for

BOOKUSD

Prepared By: Xiaomi Huang

PeckShield

May 21, 2025

Document Properties

Client	BOOKUSD
Title	Smart Contract Audit Report
Target	BOOKUSD
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 21, 2025	Xuxian Jiang	Final Release
1.0-rc1	May 18, 2025	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `BOOKUSD` 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 can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About BOOKUSD

`BOOKUSD` is a native lending protocol, allowing users to deposit `$BOOK` (a digital asset on BNB Chain) and borrow the `$BUD` stablecoin. Users can stake `$BUD` in the `Stability Pool` to earn `$BUSS`, and participate in platform fee-sharing through the `$BUSS` pool. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of BOOKUSD

Item	Description
Name	BOOKUSD
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 21, 2025

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that BOOKUSD assumes a trusted price oracle with timely market price feeds for supported assets.

- https://github.com/BookMemeBsc/bookusd-contracts.git (4f70039)

And this is the commit ID after all fixes for the issues found in the audit have been checked in.

- https://github.com/BookMemeBsc/bookusd-contracts.git (7ded570, 3bbdae2)

1.2 About PeckShield

PeckShield Inc. [9] 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			
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low
	High	*Medium*	*Low*
		Likelihood	

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [8]:

- 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

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

is considered safe regarding the check item. For any discovered issue, we might further deploy 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) [7], 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 BOOKUSD protocol. 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	0	
Medium	2	🟦🟦
Low	3	🟦🟦🟦
Informational	0	
Total	5	

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 2 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key BOOKUSD Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Avoidance of Repeated Initialization in CommunityIssuance	Security Features	Resolved
PVE-002	Low	Incorrect TroveLiquidated Event in TroveManager	Business Logic	Resolved
PVE-003	Low	Inconsistent Borrow Cap Enforcement in BorrowerOperations	Business Logic	Confirmed
PVE-004	Low	Improved Caller Validation in SortedTroves::insert()	Security Features	Confirmed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Resolved

Besides 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 Avoidance of Repeated Initialization in CommunityIssuance

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: CommunityIssuance
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

To engage the community, BOOKUSD has a core CommunityIssuance contract which contains the logic to issue $BUSS to protocol users. While reviewin the contrat logic, we notice an issue that may allow the privileged owner to repeatedly initialize the contract.

To elaborate, we show below the related setAddresses() routine. The current logic properly assigns a number of protocol contracts from the trusted user input. However, it comes to our attention that this routine allows for repeated address assignment. For improved confidence, we strongly suggest to make the initialization only once, blocking any further attempt to re-initialize the contracts. Note another contract BorrowerOperations shares the same issue.

```
    function setAddresses(
        address _lqtyTokenAddress,
        address _stabilityPoolAddress,
        uint256 _startTime
    ) external override onlyOwner {



        checkContract(_lqtyTokenAddress);
        checkContract(_stabilityPoolAddress);

        lqtyToken = ILQTYToken(_lqtyTokenAddress);
        stabilityPoolAddress = _stabilityPoolAddress;

        // When LQTYToken deployed, it should have transferred CommunityIssuance's LQTY entitlement
        // uint LQTYBalance = lqtyToken.balanceOf(address(this));
        // assert(LQTYBalance >= LQTYSupplyCap);

        deploymentTime = _startTime;

        emit LQTYTokenAddressSet(_lqtyTokenAddress);
        emit StabilityPoolAddressSet(_stabilityPoolAddress);
    }
```

Listing 3.1: CommunityIssuance::setAddresses()

Recommendation Revisit the above initialization routine to ensure it can only be called once.

Status The issue has been fixed by this commit: 7ded570.

3.2 Incorrect TroveLiquidated Event in TroveManager

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: TroveManager
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the TroveManager contract as an example. This contract has public functions that are used to manage current troves. While examining the TroveLiquidated events, we notice the emitted information needs to be improved.

Specifically, when a trove is liquidated during the recovery mode, the _liquidateRecoveryMode() routine will be invoked. By design, if there is BUD in the stability pool, the liquidation will only offset, with no redistribution, but at a capped rate of 1.1 and only if the whole debt can be liquidated. In the meantime, the remainder due to the capped rate will be claimable as collateral surplus. With that, the TroveLiquidated event needs to reflect the actual debt/collateral being liquidated. The current event logic shows the right debt amount (singleLiquidation.entireTroveDebt), but not the collateral amount (singleLiquidation.collToSendToSP). The exact collateral amount being liquidated is singleLiquidation.entireTroveColl - singleLiquidation.collSurplus (line 466).

```
442          ...
443          if ((_ICR >= MCR) && (_ICR < _TCR) && (singleLiquidation.entireTroveDebt <=
                 _LUSDInStabPool)) {
444              _movePendingTroveRewardsToActivePool(
445                  _activePool,
446                  _defaultPool,
447                  vars.pendingDebtReward,
448                  vars.pendingCollReward
449              );
450              assert(_LUSDInStabPool != 0);

452              _removeStake(_borrower);
453              singleLiquidation = _getCappedOffsetVals(
454                  singleLiquidation.entireTroveDebt,
455                  singleLiquidation.entireTroveColl,
456                  _price
457              );

459              _closeTrove(_borrower, Status.closedByLiquidation);
460              if (singleLiquidation.collSurplus > 0) {
461                  collSurplusPool.accountSurplus(_borrower, singleLiquidation.collSurplus)
                         ;
462              }

464              emit TroveLiquidated(
465                  _borrower,
466                  singleLiquidation.entireTroveDebt,
467                  singleLiquidation.collToSendToSP,
468                  TroveManagerOperation.liquidateInRecoveryMode
469              );
470              emit TroveUpdated(_borrower, 0, 0, 0, TroveManagerOperation.
                     liquidateInRecoveryMode);
471          }
```

Listing 3.2: TroveManager::_liquidateRecoveryMode()

Recommendation Properly emit the above TroveLiquidated event with the right debt/collateral amount. Note LUSDTokenAddressSet event should also be emitted in the LQTYStaking::setAddresses() function.

Status The issue has been fixed by this commit: 7ded570.

3.3 Inconsistent Borrow Cap Enforcement in BorrowerOperations

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `BorrowerOperations`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `BOOKUSD` protocol is no exception. Specifically, if we examine the `BorrowerOperations` contract, it has defined a number of protocol-wide risk parameters, such as `borrowCap` and `maxBorrowPerWallet`. In the following, we show the corresponding routines that allow for their changes.

```
function setBorrowCap(uint256 _cap) external override onlyOwner {
    borrowCap = _cap;
    emit BorrowCapChanged(_cap);
}

function setMaxBorrowPerWallet(uint256 _max) external override onlyOwner {
    maxBorrowPerWallet = _max;
    emit MaxBorrowPerWalletChanged(_max);
}
```

Listing 3.3: BorrowerOperations::setBorrowCap() and BorrowerOperations::setMaxBorrowPerWallet()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when enforcing them. Our analysis shows their enforcement can be improved. In the following, we show the code snippet when a new trove is opened.

```
function openTrove(
    uint _collateralAmount,
    uint _maxFeePercentage,
    uint _LUSDAmount,
    address _upperHint,
    address _lowerHint
) external override {
    require(block.timestamp >= startTime, "BorrowerOperations: Trove creation not
        yet enabled");
    // transfer BOOK in

    require(book.transferFrom(msg.sender, address(this), _collateralAmount), "Could
        not transfer token");

    ContractsCache memory contractsCache = ContractsCache(troveManager, activePool,
        lusdToken);
    LocalVariables_openTrove memory vars;

    vars.price = priceFeed.fetchPrice();
    bool isRecoveryMode = _checkRecoveryMode(vars.price);

    _requireValidMaxFeePercentage(_maxFeePercentage, isRecoveryMode);
    _requireTroveisNotActive(contractsCache.troveManager, msg.sender);

    vars.LUSDFee;
    vars.netDebt = _LUSDAmount;

    if (!isRecoveryMode) {
        vars.LUSDFee = _triggerBorrowingFee(
            contractsCache.troveManager,
            contractsCache.lusdToken,
            _LUSDAmount,
            _maxFeePercentage
        );
        vars.netDebt = vars.netDebt.add(vars.LUSDFee);
    }
    _requireAtLeastMinNetDebt(vars.netDebt);

    // --- Enforce borrow cap ---
    uint256 currentDebt = activePool.getLUSDDebt();
    require(
        borrowCap == 0  currentDebt.add(vars.netDebt) <= borrowCap,
        "BorrowerOperations: Borrow cap exceeded"
    );

    // --- Enforce max borrow per wallet ---
    uint256 userDebt = troveManager.getTroveDebt(msg.sender);
    if (msg.sender != owner()) {
        require(
            maxBorrowPerWallet == 0  userDebt.add(vars.netDebt) <=
                maxBorrowPerWallet,
            "BorrowerOperations: Max borrow per wallet exceeded"
        );
    }
    ...
}
```

Listing 3.4: BorrowerOperations::openTrove()

Specifically, when a new trove is opened, the enforcement of the global debt cap as well as the per-wallet debt cap should take into the consideration the `LUSD_GAS_COMPENSATION` cost associated with each new active trove. However, current enforcement does not include the `LUSD_GAS_COMPENSATION` cost.

Recommendation Improve the above logic to properly enforce the global debt cap as well as the per-wallet debt cap.

Status This issue has been confirmed.

3.4 Revisited Caller Validation in SortedTroves::insert()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: SortedTroves
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The BOOKUSD protocol has a core SortedTroves contract to maintain a sorted doubly linked list of active troves in descending order accordingly to their nominal individual collateral ratios (NICR). Our analysis shows that the key insert() operation is expected to be called only from the borrowerOperations contract.

To elaborate, we show below the related insert() routine, which has a rather straightforward logic in inserting a trove node into the list while maintaining the proper descending list based on its NICR. It comes to our attention that the caller is validated to be from either borrowerOperations or TroveManager. However, the current TroveManager logic will only call the reInsert() function to re-insert the node at a new position (based on its new NICR), not the insert() routine.

```
    function insert (address _id, uint256 _NICR, address _prevId, address _nextId)
        external override {
        ITroveManager troveManagerCached = troveManager;

        _requireCallerIsBOorTroveM(troveManagerCached);
        _insert(troveManagerCached, _id, _NICR, _prevId, _nextId);
    }
```

Listing 3.5: SortedTroves::insert()

Recommendation Revise the above caller-validating logic inside the insert() routine.

Status This issue has been confirmed.

3.5 Trust Issue of Admin Keys

- ID: PVE-0035
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the BOOKUSD protocol, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the system-wide operations (e.g., configure various settings and manage risk parameters). It also has the privilege to control or govern the flow of assets within the protocol contracts. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function setBurningBooks(address _address) public onlyOwner {
        burningBooks = _address;
    }
    ...
    function burnFrom(address _account, uint256 _amount) public onlyBurningBooks {
        _burn(_account, _amount);
    }
```

Listing 3.6: Example Privileged Operations in BookOfBinance

```
    function adminUpdateTwapDuration(uint32 _twapDuration) external onlyOwner {
        require(_twapDuration > 0, "RedeemProxy: twap duration must be greater than 0");
        twapDuration = _twapDuration;
    }

    function adminWithdrawToken(address _token) external onlyOwner {
        require(_token != address(0), "RedeemProxy: token is zero address");

        uint256 _amount = IERC20(_token).balanceOf(address(this));
        IERC20(_token).transfer(msg.sender, _amount);
    }
```

Listing 3.7: Example Privileged Operations in RedeemProxy

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changes 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 The issue has been resolved as the owner has been fully renounced.

4 Conclusion

In this audit, we have analyzed the design and implementation of the BOOKUSD protocol, which is a native lending protocol, allowing users to deposit $BOOK and borrow the $BUD stablecoin. Users can stake $BUD in the Stability Pool to earn $BUSS, and participate in platform fee-sharing through the $BUSS pool. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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

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

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