SMART CONTRACT AUDIT REPORT

for

Boltfi Vault

Prepared By: Xiaomi Huang

PeckShield

January 26, 2024

Document Properties

Client	Boltfi
Title	Smart Contract Audit Report
Target	Boltfi Vault
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 26, 2024	Xuxian Jiang	Final Release
1.0-rc	January 23, 2024	Xuxian Jiang	Release Candidate

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 `Boltfi` 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-documented and well-engineered, and it can benefit from addressing the reported issues. This document outlines our audit results.

1.1 About Boltfi

`Boltfi` is designed to be `ERC4626`-compliant vault that allows users to deposit and redeem assets at any time. In addition, these actions are queued and processed by the owner at a later date. The conversion of assets to shares and vice versa is based on the price at the time of processing, which can be updated by the owner at any time. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Boltfi Vault

Item	Description
Issuer	Boltfi
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 26, 2024

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

- https://github.com/boltfi/protocol-v1.git (5788c2d)

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

- https://github.com/boltfi/protocol-v1.git (a9c1ba8)

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 \ 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 [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 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

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
	DeltaPrimeLabs 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

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 `Boltfi` 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	1	■
Low	3	■ ■ ■
Informational	0	
Total	4	

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 medium-severity vulnerability and 3 low-severity vulnerabilities.

Table 2.1: Key Boltfi Vault Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited Share Redemption Logic in processRedeems()	Business Logic	Resolved
PVE-002	Low	Reduced Gas Cost in Token Deposit And Revert	Coding Practices	Resolved
PVE-003	Low	Suggested Withdrawal Fee Limit in updateWithdrawalFee()	Coding Practices	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Revisited Share Redemption Logic in processRedeems()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Vault`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Boltfi` protocol has a key `Vault` contract that enables users to deposit tokens to obtain vault share and later redeem share for underlying tokens. While examining the related redemption logic, we notice the current approach may be revisited.

To elaborate, we show below the related `processRedeems()` routine that redeems the user shares. It comes to our attention that the redemption logic enforces the following invariant, i.e., `require(_asset.balanceOf(address(this))== 0)` (line 135). This invariant in essence enforces the vault contract does not hold any underlying asset. While it is a reasonable design goal, it might make the calculation of input argument `total` complicated since any dust donation can easily make the requirement unmet.

```
122     function processRedeems(uint128 number, uint256 total) external onlyOwner
            onlyUpdatedPrice {
123         SafeERC20.safeTransferFrom(_asset, _msgSender(), address(this), total);
124 
125         for (uint256 i = 0; i < number; i++) {
126             PendingRedeem memory item = abi.decode(Queue.popFront(_redeemQueue), (
                    PendingRedeem));
127 
128             _burn(address(this), item.shares);
129 
130             uint256 assets = previewRedeem(item.shares);
131             SafeERC20.safeTransfer(_asset, item.receiver, assets);
132             emit Withdraw(item.caller, item.receiver, item.owner, assets, item.shares);
133         }
134 
```

```
        require(_asset.balanceOf(address(this)) == 0, "Incorrect total given"); // Avoid
            keeping assets in the contract
    }
```

Listing 3.1: `Vault::processRedeems()`

Recommendation Revise the above-mentioned invariant to avoid making the redemption unnecessarily complicated.

Status The issue has been resolved as the team confirms it is part of intended design.

3.2 Reduced Gas Cost in Token Deposit And Revert

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

Description

The core `Vault` contract implements an `ERC4626`-compliant vault that allows users to deposit and redeem assets with expected yields. In the process of examining the deposit logic, we notice the implementation may be optimized for reduced gas cost.

In the following, we show the implementation of the related `deposit()` routine. It has a rather straightforward logic in transferring the user funds to the designated `owner()` account and adding a new deposit entry into the pending deposit queue. Our analysis shows the funds are transferred twice, which may be consolidated into one for reduced gas cost, i.e., `SafeERC20.safeTransferFrom(_asset, _msgSender(), owner(), assets)` to replace the statements (lines 84 – 85).

```
83      function deposit(uint256 assets, address receiver) external virtual whenNotPaused {
84          SafeERC20.safeTransferFrom(_asset, _msgSender(), address(this), assets);
85          SafeERC20.safeTransfer(_asset, owner(), assets);

87          PendingDeposit memory item = PendingDeposit(_msgSender(), receiver, assets,
                uint32(block.timestamp));
88          Queue.pushBack(_depositQueue, abi.encode(item));
89      }
```

Listing 3.2: `Vault::deposit()`

Recommendation Revisit the above routine to optimize the asset transfers. Note the same issue is also applicable to another routine, i.e., `revertFrontDeposit()`.

Status The issue has been resolved as the team confirms it is part of intended design.

3.3 Suggested Withdrawal Fee Limit in updateWithdrawalFee()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Vault`
- 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 `Boltfi` protocol is no exception. Specifically, if we examine the `Vault` contract, it has defined a number of token-wide risk parameters, such as `withdrawalFee` and `price`. In the following, we show the corresponding routines that allow for their changes.

```
    function updatePrice(uint256 price_) external onlyOwner {
        require(price_ > 0, "Price must be greater than 0"); // Avoid causing issues with division
        price = price_;
        priceUpdatedAt = uint32(block.timestamp);
        emit PriceUpdate(price_);
    }

    function updateWithdrawalFee(uint256 withdrawalFee_) external onlyOwner {
        withdrawalFee = withdrawalFee_;
        emit WithdrawalFeeUpdate(withdrawalFee_);
    }
```

Listing 3.3: Vault::updatePrice()/updateWithdrawalFee()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. Specifically, the `updateWithdrawalFee` setter can be improved to further validate the given `withdrawalFee` falls in a reasonable range. For example, it needs to be smaller than `10 ** FEE_DECIMALS`. Otherwise, no vault users are able to withdraw their funds.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status The issue has been fixed by this commit: `a9c1ba8`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Vault
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Boltfi protocol, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the protocol-wide operations (e.g., parameter configuration and share price update). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged owner account and its related privileged accesses in current contracts.

```
    function updatePrice(uint256 price_) external onlyOwner {
        require(price_ > 0, "Price must be greater than 0"); // Avoid causing issues with division
        price = price_;
        priceUpdatedAt = uint32(block.timestamp);
        emit PriceUpdate(price_);
    }

    function updateWithdrawalFee(uint256 withdrawalFee_) external onlyOwner {
        withdrawalFee = withdrawalFee_;
        emit WithdrawalFeeUpdate(withdrawalFee_);
    }

    /// @dev Should pause all user actions (deposit, redeem)
    function pause() external onlyOwner {
        _pause();
    }

    function unpause() external onlyOwner {
        _unpause();
    }

    /// @dev No equivalent for ETH as it can't be recieve due to no fallback function
    function withdrawalToOwner(IERC20 token) external onlyOwner {
        uint256 balance = token.balanceOf(address(this));
        require(balance > 0, "Contract has no balance");
        SafeERC20.safeTransfer(token, owner(), balance);
    }
```

Listing 3.4: Example Privileged Operations in Vault

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `owner` account is a plain EOA account. 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.

Moreover, it should be noted that current contracts are to be deployed behind a proxy. And naturally, there is a need to properly manage the admin privileges as they are capable of upgrading the entire protocol implementation.

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 addressed as the team clarifies the use of a multisig.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Boltfi` protocol, which is designed to be `ERC4626`-compliant vault. The vault allows users to deposit and redeem assets at any time. In addition, these actions are queued and processed by the owner at a later date. The conversion of assets to shares and vice versa is based on the price at the time of processing, which can be updated by the owner at any time. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed

Meanwhile, 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.