BITVAULT

SMART CONTRACTS REVIEW

zokyo

August 5th 2025 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING BITVAULT

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: 0 points

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 1 High issue: 1 acknowledged = - 8 points deducted
- 1 Medium issue: 1 acknowledged = - 4 points deducted
- 3 Low issues: 2 resolved and 1 acknowledged = - 1 points deducted
- 3 Informational issues: 1 resolved and 2 acknowledged = 0 points deducted

Thus, 100 - 8 - 4 - 1 = 87

TECHNICAL SUMMARY

This document outlines the overall security of the BitVault smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the BitVault smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the BitVault team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the BitVault repository:
Repo: https://github.com/Popcorn-Limited/bvusd/commit/b2b61a0350cad2fa88d3b836f8c975667e3e18a1

Fixes - https://github.com/Popcorn-Limited/bvusd/pull/19

Contracts under the scope:

- Dependencies/MultiTokenWrapper.sol
- Dependencies/TokenWrapper.sol
- Dependencies/Whitelist.sol
- Dependencies/BoldConverter.sol
- Dependencies/HasWhitelist.sol
- PriceFeeds/PORExchangeRateFeed.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of BitVault smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

The BitVault codebase for audit consist of contracts MultiTokenWrapper.sol, TokenWrapper.sol, Whitelist.sol, BoldConverter.sol, HasWhitelist.sol, and PORExchangeRateFeed.sol.

MultiTokenWrapper contract allows owner to wrap multiple underlying tokens by depositing and whitelisting underlying tokens. Whereas TokenWrapper allows wrapping of a single underlying token. BoldConverter contract allows users to deposit underlying tokens and mint bvsd token in return. Same can be burned for underlying tokens again. PORExchangeRateFeed contract is using chainlink price feed to get token price. Whitelist contract allows owner to whitelist and blacklist users.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the BitVault team and the BitVault team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	No Token Separation Leading to Cross-Asset Withdrawal	High	Acknowledged
2	Oracle Response Not Validated in latestRoundData	Medium	Acknowledged
3	Inaccurate Token Scaling May Result in Zero Transfers	Low	Resolved
4	Functions deposit and withdraw are exposed to reentrancy	Low	Resolved
5	Removing underlying tokens can result in the locking of underlying tokens	Low	Acknowledged
6	Ensure that the fee from BoldConverter can be claimed	Informational	Acknowledged
7	Not checking for down L2 sequencer	Informational	Acknowledged
8	Unused imports	Informational	Resolved

No Token Separation Leading to Cross-Asset Withdrawal

Location: MultiTokenWrapper.sol

The deposit() and withdraw() functions allow interaction with multiple supported underlying tokens. However, the implementation lacks per token accounting, which leads to a critical flaw:

The system mints a generic 18-decimal wrapped token based on the deposited token's amount and decimal scaling but does not record which underlying token the deposit corresponds to. This allows the caller to:

- Deposit TokenA and receive wrapped tokens.
- Call withdraw() using those wrapped tokens to receive TokenB.
- There is no enforcement or tracking mechanism to ensure that withdrawals correspond to the same asset that was deposited.

Impact
This design flaw introduces a severe accounting mismatch. It allows value to be extracted from one underlying token pool after contributing a different token, leading to fund imbalances. Depending on token valuations and reserves, this could:

Recommendation:
Implement per-token accounting to ensure that deposited and withdrawn assets are correctly tracked and matched. For example:

Maintain a mapping of balances per user per token:

```
mapping(address => mapping(address => uint256)) public userDeposits; // user => token => amount
```

When minting wrapped tokens, associate them with a specific underlying asset or deploy a separate wrapper per token which is already implemented in scope.

Fix: The issue becomes irrelevant as Client stated they are not utilizing this contract in their protocol since they are using a Single Token Wrapper contract instead.

Oracle Response Not Validated in latestRoundData

Location: PORExchangeRateFeed.sol

The latestRoundData() function in the contract fetches data from an external oracle (POR_FEED.latestRoundData()), but does not validate the return values—particularly the answer, updatedAt, or answeredInRound fields:

```
(roundId, answer, startedAt, updatedAt, answeredInRound) =
POR_FEED.latestRoundData();
```

No checks are performed to ensure:

`answer` is a positive and reasonable value

`updatedAt` is recent enough to avoid using stale data

Instead, the answer is immediately used in a calculation, potentially propagating invalid or stale values into further logic.

Impact
Failing to validate the oracle's return values may result in use of outdated or incorrect pricing data, leading to mispricing or miscalculations. For example, an outdated updatedAt could be used in critical logic like collateral valuation or minting calculations, which then harms the economic facet of the system.

Recommendation:
Add appropriate validation checks af ter fetching the oracle response to ensure that the returned data is fresh, valid, and trustworthy. For example:

```
require(updatedAt >= block.timestamp - MAX_DELAY, "Stale oracle data");
```

Fix:
This feed contract is not meant to be used standalone. It will be integrated into the MorphoOracle, which has implemented Oracle response validation here (https://github.com/morpho-org/morpho-blue-oracles/blob/main/src/morpho-chainlink/libraries/ChainlinkDataFeedLib.sol#L20)

Inaccurate Token Scaling May Result in Zero Transfers

Location: TokenWrapper.sol

In the TokenWrapper.sol contract, the withdraw() function burns a specified amount of tokens from the caller and attempts to transfer the corresponding scaled amount of the underlying token:

```
uint256 scaledAmount = amount / 10 ** (18 - _underlyingDecimals);
```

However, when amount is small and _underlyingDecimals is significantly lower than 18 (e.g., 6 or 8), the computed scaledAmount may round down to zero due to integer division. In such cases, the function burns a non-zero number of wrapper tokens without transferring any underlying tokens in return.

Impact
This behavior can lead to a loss of user funds, as wrapper tokens are irreversibly burned, but the user receives no compensation in underlying tokens.

Recommendation:
Implement a check to ensure that amount is greater than a calculated minimum threshold before proceeding with the burn and transfer. Alternatively require that scaledAmount be a non-zero value. If the scaled amount would result in zero, the transaction should revert with a clear error message.

Functions deposit and withdraw exposed to reentrancy

Location: BoldConverter.sol

The `deposit()` and `withdraw()` functions in the contract perform external calls to user-supplied tokens `(SafeERC20.safeTransferFrom)` both before and after state-changing logic (minting and burning of bvUSD). However, these functions are not protected by a reentrancy guard `( nonReentrant modifier )`.

```
bvUSD.burn(msg.sender, amount);
...
SafeERC20.safeTransferFrom(underlying, path.underlyingReceiver, receiver, underlyingOut);
```

Because these functions involve external calls to arbitrary ERC20 tokens (which may have arbitrary implementations on reception), the contract is susceptible to reentrancy.

Impact
While there is no immediate user fund loss evident in the current logic, the presence of unguarded external calls introduces a latent security risk. Attackers could exploit this gap to manipulate the control flow as it leaves the contract open to future vulnerabilities.

Recommendation:
Apply a reentrancy guard (e.g., nonReentrant from ReentrancyGuard) to both the `deposit()` and `withdraw()` functions to prevent recursive calls during external token interactions.

Removing underlying tokens can result in the locking of underlying tokens

In Contract MultiTokenWrapper.sol, the method removeUnderlying removes the underlying tokens from the mapping. This does not check if the contract has any pending underlying tokens to be withdrawn.

In case there are underlying tokens that have not been withdrawn yet, they will be stuck forever, as the withdraw(...) method first checks if the underlying token is whitelisted or not.

The same issue lies in the contract BoldConverter when a path is removed for a particular underlying token, but that underlying yet not been withdrawn.

Recommendation:
Update the contract logic to have an internal mapping for the total amount deposited per underlying token. Consider doing it for both the contracts, i.e MultiTokenWrapper.sol and BoldConverter.sol

mapping(address ⇒ uint256) totalDeposited;

Increment it when the user deposits for an underlying token.
totalDeposited[underlying] += amount;

Decrement is when the user withdraws for an underlying token.
totalDeposited[underlying] -= amount;

Remove an underlying token only when` totalDeposited[underlying] == 0`

INFORMATIONAL-1 | ACKNOWLEDGED

Ensure that the fee from BoldConverter can be claimed

In Contract BoldConverter, a fee is deducted when user withdraws.

```
// scale amount and subtract fee
underlyingOut =
    withdrawalAmount -
    ((withdrawalAmount * path.withdrawalFee) / MAX_FEE);
```

Here, the fee is an amount that will not be transferred to the user but will remain in the path.underlyingReceiver contract for the owner/admin to claim.

Recommendation:
Ensure that the fee can be claimed by the owner/admin and not stuck in the path.underlyingReceiver contract.

INFORMATIONAL-2 | ACKNOWLEDGED

Not checking for down L2 sequencer

In the Contract PoreExchangeRateFeed.sol, the method latestRoundData(...) calls priceFeed.latestRoundData(...) to get the latest price for assets. If this oracle contract is used on the L2 layer, it is important to check if the sequencer is down or not to avoid stale pricing data that appears fresh.

The official Chainlink documentation for the same is provided here.

Recommendation:
Use the sequencer oracle to determine whether the sequencer is offline or not, and don't allow orders to be executed while the sequencer is offline.

Unused imports

In Contract BoldConverter.sol, the following import is not used.

```
import {ERC20} from
"openzeppelin-contracts/contracts/token/ERC20/ERC20.sol";
```

Recommendation:
Remove unused imports.

	Dependencies/MultiTokenWrapper.sol Dependencies/TokenWrapper.sol Dependencies/Whitelist.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	Dependencies/BoldConverter.sol Dependencies/HasWhitelist.sol PriceFeeds/PORExchangeRateFeed.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the BitVault team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the BitVault team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo