SMART CONTRACT AUDIT REPORT

for

Pluz Protocol

Prepared By: Xiaomi Huang

PeckShield

August 18, 2024

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	August 18, 2024	Xuxian Jiang	Final Release
1.0-rc1	August 10, 2024	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 Pluz 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 Pluz

Pluz is a permissionless lending protocol that allows users to have a leveraged access on their collateral to use in the most well vetted and popular DApps. The protocol is designed to help users have an optimized yield and point farming by aiming to capitalize on DeFi protocols yields, rewards, and points. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Pluz

Item	Description
Target	Pluz
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	August 18, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit covers all contracts in the repository, except the following one: `GammaNarrowUniswapV3Strategy`.

- https://github.com/pluzfi/pluz.git (68f3217)

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

- https://github.com/pluzfi/pluz.git (b8648fe)

1.2 About PeckShield

PeckShield Inc. [12] 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 OWASP Risk Rating Methodology [11]:

- 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 list of check 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

Table 1.3: The Full List of Check Items

Category	Check Item
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

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) [10], 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.

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 Logics	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 `Pluz` 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	1	■
Medium	2	■ ■
Low	3	■ ■ ■
Informational	0	
Total	6	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 2 medium-severity vulnerabilities, and 3 low-severity vulnerabilities.

Table 2.1: Key Pluz Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improper withdraw() Logic in LendingPool	Business Logic	Resolved
PVE-002	Medium	Improved Liquidation Logic in PluzAccountManager	Business Logic	Resolved
PVE-003	Low	Improved Precision in ERC20CollateralVault::previewDeposit()	Numeric Errors	Resolved
PVE-004	Low	Suggested Caller Validation in StrategyVault::liquidate()	Coding Practices	Resolved
PVE-005	Low	Explicit Enforcement of Implicit Assumption in PluzAccountManager	Coding Practices	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Improper withdraw() Logic in LendingPool

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `LendingPool`
- Category: Business Logic [8]
- CWE subcategory: CWE-770 [4]

Description

`Pluz` is a permissionless lending protocol that allows users to have a leveraged access on their collateral. The lending pool is implemented in a core contract named `LendingPool`. While examining the lending support, we notice current implementation on collateral withdrawal and debt repayment can be improved.

In the following, we show the implementation of the related `withdraw()` routine. This routine allows an user to withdraw his or her collateral. However, it comes to our attention that the actual asset amount for withdrawal (`convertAmount`) may be prematurely computed as `_convertAmount (amountToWithdraw, IERC20Rebasing(address(reserve.asset)))` (line 233). The reason is that the given variable of `amountToWithdraw` may be later updated when the input `amount` is larger than the user balance (line 241). With that, there is a need to compute the actual asset amount `convertAmount` after the `amountToWithdraw` is finalized (line 242).

```
230     function withdraw(uint256 amount) public virtual whenNotPaused nonReentrant returns
            (uint256) {
231         uint256 amountToWithdraw = amount;

233         uint256 convertAmount = _convertAmount(amountToWithdraw, IERC20Rebasing(address(
                reserve.asset)));

235         _beforeAction();

237         bool isMaxWithdraw = false;
```

```
        uint256 userBalance = liquidityToken.balanceOf(msg.sender);

        if (amount >= userBalance) {
            amountToWithdraw = userBalance;
            isMaxWithdraw = true;
        }

        reserve.assetBalance -= amountToWithdraw;

        liquidityToken.burn(msg.sender, amountToWithdraw, reserve.liquidityIndex,
            isMaxWithdraw, MathUtils.ROUNDING.UP);
        IERC20Rebasing(address(reserve.asset)).unwrap(amountToWithdraw);
        _actualAsset.safeTransfer(msg.sender, convertAmount);

        _mintToTreasury();
        _updateInterestRate();

        emit Withdraw(msg.sender, amountToWithdraw);
        return amountToWithdraw;
    }
```

Listing 3.1: `LendingPool::withdraw()`

Recommendation Improve the above-mentioned routine to properly compute the actual asset amount for withdrawal. Note the same issue is also applicable to the `repay()` routine from the same contract.

Status This issue has been fixed in the following commit: `b8648fe`.

3.2 Improved Liquidation Logic in PluzAccountManager

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `PluzAccountManager`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `Pluz` protocol has a core `PluzAccountManager` contract to oversee the account creation and management. In the process of examining the liquidation logic of an underwater position, we notice current implementation should be improved.

In the following, we show the implementation of the related routine, i.e., `liquidateCollateral()`. This routine has a rather straightforward logic in repaying the debt from the calling user (a.k.a., liquidator) and seizing the collateral from the borrower being liquidated. It comes to our attention that the `_lendAsset` token is a rebasing one and the repayment funds are directly transferred from the liquidator in term of `lendPoolActualAsset` (line 271). With that, there is no need to unwrap the `_lendAsset` at all [1] (line 270).

Moreover, this routine can also be improved to have the `nonReentrant` modifier to block unintended reentrancy attempts. Note other public functions from this `PluzAccountManager` contract have this `nonReentrant` modifier consistently applied.

```
245     function liquidateCollateral(address account, uint256 debtToCover, address
            liquidationFeeTo) public {
246         AccountLib.Health memory health = getAccountHealth(account);

248         if (!health.isLiquidatable) revert Errors.AccountHealthy();

250         // Mark account as liquidatable if it isn't already.
251         if (_accountLiquidationStartTime[account] == 0) {
252             _accountLiquidationStartTime[account] = block.timestamp;
253             emit AccountLiquidationStarted(account);
254             this._afterLiquidationStarted(account);
255         }

257         // The collateral is credited to the owner of the Account, not the Account
                itself.
258         address accountOwner = _accountOwnerCache[account];
259         uint256 debtAmount = getDebtAmount(account);

261         AccountLib.CollateralLiquidation memory _result =
262             _simulateCollateralLiquidation(accountOwner, debtAmount, debtToCover);

264         // Transfer collateral to caller and their fee wallet
265         _withdrawAssets(accountOwner, msg.sender, _result.collateralAmount - _result.
                bonusCollateral);
266         _withdrawAssets(accountOwner, liquidationFeeTo, _result.bonusCollateral);

268         // Transfer debt from sender to account.
269         uint256 convertAmount = _convertAmount(_result.actualDebtToLiquidate,
                IERC20Rebasing(address(_lendAsset)));
270         IERC20Rebasing(address(_lendAsset)).unwrap(_result.actualDebtToLiquidate);
271         _lendPoolActualAsset.safeTransferFrom(msg.sender, account, convertAmount);
272         IAccount(account).repay(_result.actualDebtToLiquidate);

274         emit CollateralLiquidation(
275             account, _result.collateralAmount, _result.bonusCollateral, _result.
                    actualDebtToLiquidate
276         );
277     }
```

Listing 3.2: `PluzAccountManager::liquidateCollateral()`

[1]In fact, this unwrap call is likely to revert.

Recommendation Revise the above-mentioned routine to properly liquidate an underwater user position.

Status This issue has been fixed in the following commit: `b8648fe`.

3.3 Improved Precision in ERC20CollateralVault::previewDeposit()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ERC20CollateralVault`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [2]

Description

As mentioned in Section 3.2, the `Pluz` protocol has a core `PluzAccountManager` contract to oversee the account creation and management. This contract also acts as the vault that holds the user collateral. In the process of examining the collateral-adding logic, we notice current implementation has a precision issue that can be improved.

To elaborate, we show below the related `previewDeposit()` routine. As the name indicates, this routine is designed to simulate the effects of a user deposit at the current block and compute the actual deposit amount and the resulting vault share. However, the actual deposit amount should be the given assets and the resulting share should be computed in a manner that favors the protocol. Our analysis shows the resulting share is properly computed while the actual deposit amount needs to be the given `assets`, instead of current `_convertToAssets(shares)` (line 74).

```
72     function previewDeposit(uint256 assets) public view virtual returns (uint256
           updatedAssets, uint256 shares) {
73         shares = _convertToShares(assets);
74         updatedAssets = _convertToAssets(shares);
75     }
```

Listing 3.3: `ERC20CollateralVault::previewDeposit()`

Recommendation Properly revise the above routine to ensure the precision loss needs to be computed in favor of the protocol, instead of the user.

Status This issue has been fixed in the following commit: `b8648fe`.

3.4 Suggested Caller Validation in StrategyVault::liquidate()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: StrategyVault
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

The Pluz protocol has a standalone StrategyVault contract to hold the investment assets. When the user funds (including collateral and investment assets) are insufficient to cover the borrowed debt, the user position may subject to liquidation. Our analysis on the liquidation logic indicates the implementation may be improved with better caller validation.

To elaborate, we show below the code snippet of the related liquidate() routine from the StrategyVault contract. This routine has an implicit assumption that the caller is an IAccount, which should be explicitly validated. Specifically, we can evaluate the queried manager is the protocol-wide account manager in charge and the calling user is a legitimate account instantiated by the manager.

```
function liquidate(
    address receiver,
    uint256,
    bytes memory data
)
    external
    payable
    virtual
    returns (uint256 receivedAssets)
{
    // We assume `msg.sender` is an IAccount because only AccountManagers can
        deposit into Strategies
    // on behalf of Accounts.

    uint256 totalShares = balanceOf(msg.sender);

    // Get the Manager that created the Account to check the Account's liquidation
        status
    IAccountManager manager = IAccount(msg.sender).getManager();
    AccountLib.LiquidationStatus memory status = manager.getAccountLiquidationStatus
        (msg.sender);

    if (!status.isLiquidating) {
        revert Errors.AccountNotBeingLiquidated();
    }
    ..
```

```
    }
```

Listing 3.4: `StrategyVault::liquidate()`

Recommendation Improve the above routine to ensure the calling user is a legitimate account instantiated by the protocol-wide account manager contract.

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

3.5 Explicit Enforcement of Implicit Assumption in PluzAccountManager

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `PluzAccountManager`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, `Pluz` is a permissionless lending protocol that supports a number of lending assets. In the process of examining the way lending assets are supported, we notice the implicit assumptions inherent in current implementation. These implicit assumptions are suggested to be removed by explicitly adding necessary requirements.

If we use the `PluzAccountManager` contract as an example, it has an implicit assumption that the debt asset and collateral asset should have the same decimals and have 18 decimals. However, this implicit assumption should be replaced with explicit `require` statements. In addition, the `StrategyVault` contract also assumes the underlying token precision is fixed to be 18 decimals (line 25), which is better explicitly enforced as well.

```
25 /// @dev Token precision is fixed to be 18 decimals.
26 abstract contract StrategyVault is IStrategyVault, Context, ERC20, ProtocolModule,
       Pausable, StrategyVaultEvents {
27     using SafeERC20 for IERC20;
28     using FixedPointMathLib for uint256;
29     ...
30 }
```

Listing 3.5: The `StrategyVault` Contract

Recommendation Remove the above implicit assumptions with explicit `require` statements.

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

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `Pluz` protocol, there is a privileged account `owner` that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and role assignment). 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 account and related privileged accesses in current contracts.

```
    function setMinimumOpenBorrow(uint256 minimumOpenBorrow) external onlyOwner {
        _minimumOpenBorrow = minimumOpenBorrow;
    }

    function updateLenderStatus(address lender, bool status) external virtual override {
     }

    function setDepositCap(uint256 newDepositCap) external onlyOwner {
        depositCap = newDepositCap;
        emit DepositCapUpdated(newDepositCap);
    }

    /// @notice Let the owner pause deposits and borrows
    function pause() external onlyOwner {
        _pause();
    }

    /// @notice Let the owner unpause deposits and borrows
    function unpause() external onlyOwner {
        _unpause();
    }
```

Listing 3.6: Example Privileged Functions in `LendingPool`

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 Make the privileges explicit to the protocol users.

Status This issue has been mitigated as the team confirms the use of a multi-sig to manage the admin privilege.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Pluz` protocol, which is a permissionless lending protocol by allowing users to have a leveraged access on their collateral to use in the most well vetted and popular DApps. The protocol is designed to help users have an optimized yield and point farming by aiming to capitalize on DeFi protocols yields, rewards, and points. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[4] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

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

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

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

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

[9] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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