SMART CONTRACT AUDIT REPORT

for

U235 Finance

Prepared By: Xiaomi Huang

PeckShield

February 25, 2024

Document Properties

Client	U235 Finance
Title	Smart Contract Audit Report
Target	U235 Finance
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	February 25, 2024	Xuxian Jiang	Final Release
1.0-rc	February 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 source code of the `U235 Finance` protocol, we outline in this 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 contract can be further improved due to the presence of the identified issues. This document outlines our audit results.

1.1 About U235 Finance

`U235 Finance` is the next-generation decentralized lending protocol on `Scroll` network. Based on `AaveV3`, it is non-custodial, permissionless, secure, and incorporates cutting-edge `DeFi` mechanisms and solutions to offer users a flexible and highly customizable `DeFi` lending experience. Built-in mechanisms and rewards incentivize participation, and innovative tokenomics ensure unparalleled sustainability. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of U235 Finance

Item	Description
Name	U235 Finance
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 25, 2024

In the following, we show the Git repository of reviewed file and the commit hash value used in this audit. Note the audited smart contracts are forked from the popular `AaveV3` protocol with the `v1.19.2` release. Note the stable rate borrowing feature should be disabled from the deployment.

- https://github.com/L2X-pro/contracts-internal.git (c62e96e)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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 deploy contract on our private testnet and run tests to confirm the findings. If necessary, we would additionally build

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

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 contract 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 contract 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 contract from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [5], 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 contract, 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 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 `U235 Finance` 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 logics, 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	0	
Informational	1	■
Total	2	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of. 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, this smart contract is 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 1 informational issue.

Table 2.1: Key U235 Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Possible Precision Issue in SupplyLogic::executeWithdraw()	Numeric Errors	Resolved
PVE-002	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 contract is being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Possible Precision Issue in SupplyLogic::executeWithdraw()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SupplyLogic`
- Category: Numeric Errors [4]
- CWE subcategory: CWE-190 [1]

Description

As mentioned earlier, the `U235 Finance` protocol is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities. While reviewing the withdrawal logic, we notice the current implementation has a rounding issue, which may be exploited under an empty market situation.

To elaborate, we show below the related `burn()/_burnScaled()` routines. The first routine is used to redeem the collateral while the second routine updates the user share as well as the total supply. It comes to our attention that the burn share amount might be computed by favoring the withdrawing user, which allows a malicious actor to manipulate an empty market and steal funds from other markets.[1]

```
function burn(
  address from,
  address receiverOfUnderlying,
  uint256 amount,
  uint256 index
) external virtual override onlyPool {
  _burnScaled(from, receiverOfUnderlying, amount, index);
  if (receiverOfUnderlying != address(this)) {
    IERC20(_underlyingAsset).safeTransfer(receiverOfUnderlying, amount);
  }
```

[1] A delicate scenario has been prepared and shared separately to the protocol team.

```
}
```

Listing 3.1: `AToken::burn()`

```
function _burnScaled(address user, address target, uint256 amount, uint256 index)
    internal {
  uint256 amountScaled = amount.rayDiv(index);
  require(amountScaled != 0, Errors.INVALID_BURN_AMOUNT);

  uint256 scaledBalance = super.balanceOf(user);
  uint256 balanceIncrease = scaledBalance.rayMul(index) -
    scaledBalance.rayMul(_userState[user].additionalData);

  _userState[user].additionalData = index.toUint128();

  _burn(user, amountScaled.toUint128());
  ...
}
```

Listing 3.2: `ScaledBalanceTokenBase::_burnScaled()`

Recommendation Revisit the above routine to properly ensure the above empty market situation can be safely avoided.

Status This issue has been resolved by following the above suggestion.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the `U235 Finance` protocol, there are a number of privileged account that play a critical role in governing and regulating the system-wide operations (e.g., configure various parameters, adjust fee, support/freeze markets, and execute other privileged operations). They also have the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged accounts need to be scrutinized. In the following, we examine the privileged accounts and the related privileged accesses in current contracts.

```
function setBorrowCap(
  address asset,
  uint256 newBorrowCap
```

```
) external override onlyRiskOrPoolAdmins {
  DataTypes.ReserveConfigurationMap memory currentConfig = _pool.getConfiguration(
      asset);
  uint256 oldBorrowCap = currentConfig.getBorrowCap();
  currentConfig.setBorrowCap(newBorrowCap);
  _pool.setConfiguration(asset, currentConfig);
  emit BorrowCapChanged(asset, oldBorrowCap, newBorrowCap);
}

/// @inheritdoc IPoolConfigurator
function setSupplyCap(
  address asset,
  uint256 newSupplyCap
) external override onlyRiskOrPoolAdmins {
  DataTypes.ReserveConfigurationMap memory currentConfig = _pool.getConfiguration(
      asset);
  uint256 oldSupplyCap = currentConfig.getSupplyCap();
  currentConfig.setSupplyCap(newSupplyCap);
  _pool.setConfiguration(asset, currentConfig);
  emit SupplyCapChanged(asset, oldSupplyCap, newSupplyCap);
}

/// @inheritdoc IPoolConfigurator
function setLiquidationProtocolFee(
  address asset,
  uint256 newFee
) external override onlyRiskOrPoolAdmins {
  require(newFee <= PercentageMath.PERCENTAGE_FACTOR, Errors.
      INVALID_LIQUIDATION_PROTOCOL_FEE);
  DataTypes.ReserveConfigurationMap memory currentConfig = _pool.getConfiguration(
      asset);
  uint256 oldFee = currentConfig.getLiquidationProtocolFee();
  currentConfig.setLiquidationProtocolFee(newFee);
  _pool.setConfiguration(asset, currentConfig);
  emit LiquidationProtocolFeeChanged(asset, oldFee, newFee);
}

/// @inheritdoc IPoolConfigurator
function setEModeCategory(
  uint8 categoryId,
  uint16 ltv,
  uint16 liquidationThreshold,
  uint16 liquidationBonus,
  address oracle,
  string calldata label
) external override onlyRiskOrPoolAdmins {
  require(ltv != 0, Errors.INVALID_EMODE_CATEGORY_PARAMS);
  require(liquidationThreshold != 0, Errors.INVALID_EMODE_CATEGORY_PARAMS);

  // validation of the parameters: the LTV can
  // only be lower or equal than the liquidation threshold
  // (otherwise a loan against the asset would cause instantaneous liquidation)
```

```
    require(ltv <= liquidationThreshold, Errors.INVALID_EMODE_CATEGORY_PARAMS);
    require(
      liquidationBonus > PercentageMath.PERCENTAGE_FACTOR,
      Errors.INVALID_EMODE_CATEGORY_PARAMS
    );

    // if threshold * bonus is less than PERCENTAGE_FACTOR, it's guaranteed that at the moment
    // a loan is taken there is enough collateral available to cover the liquidation bonus
    require(
      uint256(liquidationThreshold).percentMul(liquidationBonus) <=
        PercentageMath.PERCENTAGE_FACTOR,
      Errors.INVALID_EMODE_CATEGORY_PARAMS
    );

    address[] memory reserves = _pool.getReservesList();
    for (uint256 i = 0; i < reserves.length; i++) {
      DataTypes.ReserveConfigurationMap memory currentConfig = _pool.getConfiguration(reserves[i]);
      if (categoryId == currentConfig.getEModeCategory()) {
        require(ltv > currentConfig.getLtv(), Errors.INVALID_EMODE_CATEGORY_PARAMS);
        require(
          liquidationThreshold > currentConfig.getLiquidationThreshold(),
          Errors.INVALID_EMODE_CATEGORY_PARAMS
        );
      }
    }

    _pool.configureEModeCategory(
      categoryId,
      DataTypes.EModeCategory({
        ltv: ltv,
        liquidationThreshold: liquidationThreshold,
        liquidationBonus: liquidationBonus,
        priceSource: oracle,
        label: label
      })
    );
    emit EModeCategoryAdded(categoryId, ltv, liquidationThreshold, liquidationBonus, oracle, label);
  }

  /// @inheritdoc IPoolConfigurator
  function setAssetEModeCategory(
    address asset,
    uint8 newCategoryId
  ) external override onlyRiskOrPoolAdmins {
    DataTypes.ReserveConfigurationMap memory currentConfig = _pool.getConfiguration(asset);

    if (newCategoryId != 0) {
```

```
      DataTypes.EModeCategory memory categoryData = _pool.getEModeCategoryData(newCategoryId);
      require(
        categoryData.liquidationThreshold > currentConfig.getLiquidationThreshold(),
        Errors.INVALID_EMODE_CATEGORY_ASSIGNMENT
      );
    }
    uint256 oldCategoryId = currentConfig.getEModeCategory();
    currentConfig.setEModeCategory(newCategoryId);
    _pool.setConfiguration(asset, currentConfig);
    emit EModeAssetCategoryChanged(asset, uint8(oldCategoryId), newCategoryId);
  }

  /// @inheritdoc IPoolConfigurator
  function setUnbackedMintCap(
    address asset,
    uint256 newUnbackedMintCap
  ) external override onlyRiskOrPoolAdmins {
    DataTypes.ReserveConfigurationMap memory currentConfig = _pool.getConfiguration(asset);
    uint256 oldUnbackedMintCap = currentConfig.getUnbackedMintCap();
    currentConfig.setUnbackedMintCap(newUnbackedMintCap);
    _pool.setConfiguration(asset, currentConfig);
    emit UnbackedMintCapChanged(asset, oldUnbackedMintCap, newUnbackedMintCap);
  }
```

Listing 3.3: Example Privileged Functions in `PoolConfigurator`

Note that if these privileged accounts are plain EOA accounts, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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 mitigated as the team makes use of a multisig to act as the privileged owner.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `U235 Finance` protocol, which is the next-generation decentralized lending protocol on `Scroll` network. Based on `AaveV3`, it is non-custodial, permissionless, secure, and incorporates cutting-edge `DeFi` mechanisms and solutions to offer users a flexible and highly customizable `DeFi` lending experience. Built-in mechanisms and rewards incentivize participation, and innovative tokenomics ensure unparalleled sustainability. The current code base is well organized and those identified issues are promptly confirmed and addressed.

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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

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

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

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

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

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