SMART CONTRACT AUDIT REPORT

for

UWU

Prepared By: Xiaomi Huang

PeckShield

December 31, 2022

Document Properties

Client	UWU
Title	Smart Contract Audit Report
Target	UWU
Version	1.0
Author	Stephen Bie
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 31, 2022	Stephen Bie	Final Release
1.0-rc	November 2, 2022	Stephen Bie	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the UWU 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 UWU

UWU is a decentralized non-custodial liquidity markets protocol that is developed on top of one of the largest DeFi protocols, i.e., AAVE. The UWU protocol allows users to participate as depositors or borrowers. Depositors provide liquidity to the market to earn passive income, while borrowers are able to borrow in an over-collateralized (perpetually) or under-collateralized (one-block liquidity) fashion. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of UWU

Item	Description
Target	UWU
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	December 31, 2022

In the following, we show the audited contracts deployed at the Ethereum chain. Note that UWU assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- MultiFeeDistribution.sol, StakingRewards.sol, ChefIncentivesController.sol, Leverager.sol, `ReserveLogic.sol`, `GenericLogic.sol`, `ValidationLogic.sol`, `LendingPool.sol`, `LendingPoolConfigurator.sol`, `AToken.sol`, `VariableDebtToken.sol`, and `AaveOracle.sol`.

Additionally, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that this audit only covers the `fallback-oracle/FallbackOracle.sol` contract.

- https://github.com/UwU-Lend/uwu-contracts.git (7bf701c)

And these are the new deployed contracts after all fixes have been checked in:

- `MultiFeeDistributionV2.sol` and `Leverager.sol`.

1.2 About PeckShield

PeckShield Inc. [8] 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

Impact / **Likelihood**

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- 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 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) [6], 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

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

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.

bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

PeckShield

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `UWU` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	3	■■■
Undetermined	1	■
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 that 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 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key UWU Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Undetermined	Potential Protocol Risk from Low-Liquidity Assets	Business Logic	Confirmed
PVE-002	Low	Fork-Compliant Domain Separator in AToken	Business Logic	Confirmed
PVE-003	Low	Flashloan-assisted Lowered Stable-BorrowRate for Mode-Switching Users	Time and State	Confirmed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-006	Medium	Revisited Logic of MultiFeeDistribution::withdraw()	Business Logic	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Potential Protocol Risk from Low-Liquidity Assets

- ID: PVE-001
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: UWU Protocol
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

With the occurrence of the `Mango Market Incident` on `Solana`, the risk of liquidity attacks on lending platforms attracts much attention from the entire DeFi community. In the following, we give one possible vector to illustrate the risk. A malicious actor may borrow a large amount of the available supply of a token (such as `ZRX`) from the lending market and sell it across multiple centralized and decentralized exchanges to depress the open market price. Once the price oracle of the lending market is updated with a lower price, the malicious actor may then withdraw most of the original collateral.

To elaborate, the malicious actor supplies `$30M` stablecoins as collateral firstly (Step I), secondly borrows `$20M` illiquid token (Step II), next sells it to depress the token's market price by `95%` and realizes `$7.5M` (Step III), and finally withdraws `$28M` collateral with the user's debt going down to `$1M` (Step IV). Overall, the malicious actor profits `$5.5M` leaving lending market with bad debt. The `Market Manipulation Risk` report [9] shows more details.

Our analysis shows that the `UWU` protocol supports a few tokens (e.g., `SifuM/Sifu/sSPELL/wMEMO`), which expose the similar `market manipulation risk`.

Recommendation Evaluate the current set of assets supported in `UWU` and revisit possible risks from low-liquidity ones to avoid the above `market manipulation risk`.

Status The issue has been confirmed by the team. The team has used price range validation to mitigate the issue.

3.2 Fork-Compliant Domain Separator in AToken

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: AToken
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The AToken token contract strictly follows the widely-accepted ERC20 specification. In the meantime, we notice the support of EIP-2612 with the permit() function that allows for approvals to be made via secp256k1 signatures. Interestingly, we notice the state variable DOMAIN_SEPARATOR is initialized once inside the initialize() function (lines 79-87).

```
62      function initialize(
63          ILendingPool pool,
64          address treasury,
65          address underlyingAsset,
66          IAaveIncentivesController incentivesController,
67          uint8 aTokenDecimals,
68          string calldata aTokenName,
69          string calldata aTokenSymbol,
70          bytes calldata params
71      ) external override initializer {
72          uint256 chainId;
73
74          //solium-disable-next-line
75          assembly {
76              chainId := chainid()
77          }
78
79          DOMAIN_SEPARATOR = keccak256(
80              abi.encode(
81                  EIP712_DOMAIN,
82                  keccak256(bytes(aTokenName)),
83                  keccak256(EIP712_REVISION),
84                  chainId,
85                  address(this)
86              )
87          );
88
89          ...
90      }
```

Listing 3.1: AToken::initialize()

The DOMAIN_SEPARATOR is used in the permit() function and should be unique to the contract and chain in order to prevent replay attacks from other domains. However, when analyzing this `permit()` routine, we realize the current implementation needs to be improved by recalculating the value of `DOMAIN_SEPARATOR` inside the `permit()` function, for the very purpose of preventing cross-chain replay attacks. Specifically, when there is a chain-level hard-fork, because of the pre-computed `DOMAIN_SEPARATOR`, a valid signature for one chain could be replayed on the other.

```
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external {
        require(owner != address(0), 'INVALID_OWNER');
        //solium-disable-next-line
        require(block.timestamp <= deadline, 'INVALID_EXPIRATION');
        uint256 currentValidNonce = _nonces[owner];
        bytes32 digest =
        keccak256(
            abi.encodePacked(
            '\x19\x01',
            DOMAIN_SEPARATOR,
            keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value,
                currentValidNonce, deadline))
            )
        );
        require(owner == ecrecover(digest, v, r, s), 'INVALID_SIGNATURE');
        _nonces[owner] = currentValidNonce.add(1);
        _approve(owner, spender, value);
    }
```

Listing 3.2: `AToken::permit()`

Recommendation Recalculate the value of `DOMAIN_SEPARATOR` inside the `permit()` function.

Status The issue has been confirmed by the team.

3.3 Flashloan-assisted Lowered StableBorrowRate for Mode-Switching Users

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: LendingPool
- Category: Business Logic [5]
- CWE subcategory: CWE-837 [2]

Description

By design, the UWU protocol supports both variable and stable borrow rates. The variable borrow rate follows closely the market dynamics and can be changed on each user interaction (either borrow, deposit, withdraw, repayment or liquidation). The stable borrow rate instead will be unaffected by these actions. However, implementing a fixed stable borrow rate model on top of a dynamic reserve pool is complicated and the protocol provides the rate-rebalancing support to work around dynamic changes in market conditions or increased cost of money within the pool.

In the following, we show the code snippet of `swapBorrowRateMode()` which allows users to swap between stable and variable borrow rate modes. It follows the same sequence of convention by firstly validating the inputs (Step I), secondly updating relevant reserve states (Step II), then switching the requested borrow rates (Step III), next calculating the latest interest rates (Step IV), and finally performing external interactions, if any (Step V).

```
    function swapBorrowRateMode(address asset, uint256 rateMode) external override
        whenNotPaused {
        DataTypes.ReserveData storage reserve = _reserves[asset];

        (uint256 stableDebt, uint256 variableDebt) = Helpers.getUserCurrentDebt(msg.
            sender, reserve);

        DataTypes.InterestRateMode interestRateMode = DataTypes.InterestRateMode(
            rateMode);

        ValidationLogic.validateSwapRateMode(
            reserve,
            _usersConfig[msg.sender],
            stableDebt,
            variableDebt,
            interestRateMode
        );

        reserve.updateState();

        if (interestRateMode == DataTypes.InterestRateMode.STABLE) {
```

```
            IStableDebtToken(reserve.stableDebtTokenAddress).burn(msg.sender, stableDebt);
            IVariableDebtToken(reserve.variableDebtTokenAddress).mint(
                msg.sender,
                msg.sender,
                stableDebt,
                reserve.variableBorrowIndex
            );
        } else {
            IVariableDebtToken(reserve.variableDebtTokenAddress).burn(
                msg.sender,
                variableDebt,
                reserve.variableBorrowIndex
            );
            IStableDebtToken(reserve.stableDebtTokenAddress).mint(
                msg.sender,
                msg.sender,
                variableDebt,
                reserve.currentStableBorrowRate
            );
        }

        reserve.updateInterestRates(asset, reserve.aTokenAddress, 0, 0);

        emit Swap(asset, msg.sender, rateMode);
    }
```

Listing 3.3: `LendingPool::swapBorrowRateMode()`

Our analysis shows this `swapBorrowRateMode()` routine can be affected by a flashloan-assisted sandwiching attack such that the new stable borrow rate becomes the lowest possible. Note this attack is applicable when the borrow rate is switched from variable to stable rate. Specifically, to perform the attack, a malicious actor can first request a flashloan to deposit into the reserve pool so that the reserve's utilization rate is close to 0, then invoke `swapBorrowRateMode()` to perform the variable-to-borrow rate switch and enjoy the lowest `currentStableBorrowRate` (thanks to the nearly 0 utilization rate in current reserve), and finally withdraw to return the flashloan. A similar approach can also be applied to bypass `maxStableLoanPercent` enforcement in `validateBorrow()`.

Recommendation Revise the current implementation to defensively detect sudden changes to a reserve utilization and block malicious attempts.

Status The issue has been confirmed be the team.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Leverager`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transferFrom()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. Specifically, the `transferFrom()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transferFrom()` interface with a `bool` return value. As a result, the call to `transferFrom()` may expect a return value. With the lack of return value of `USDT`'s `transferFrom()`, the call will be unfortunately reverted.

```
    function transferFrom(address _from, address _to, uint _value) public onlyPayloadSize(3 * 32) {
        var _allowance = allowed[_from][msg.sender];

        // Check is not needed because sub(_allowance, _value) will already throw if this condition is not met
        // if (_value > _allowance) throw;

        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        if (_allowance < MAX_UINT) {
            allowed[_from][msg.sender] = _allowance.sub(_value);
        }
        uint sendAmount = _value.sub(fee);
        balances[_from] = balances[_from].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(_from, owner, fee);
        }
        Transfer(_from, _to, sendAmount);
    }
```

Listing 3.4: `USDT Token Contract`

Because of that, a normal call to `transferFrom()` is suggested to use the safe version, i.e., `safeTransferFrom()`. In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()` as well, i.e., `safeApprove()`.

In the following, we show the `loop()` routine in the `Leverager` contract. If the `USDT` token is supported as `asset`, the unsafe version of `IERC20(asset).transferFrom(msg.sender, address(this), amount)` (line 64) may revert as there is no return value in the `USDT` token contract's `transferFrom()` implementation (but the `IERC20` interface expects a return value).

```
56      function loop(
57          address asset,
58          uint256 amount,
59          uint256 interestRateMode,
60          uint256 borrowRatio,
61          uint256 loopCount
62      ) external {
63          uint16 referralCode = 0;
64          IERC20(asset).transferFrom(msg.sender, address(this), amount);
65          IERC20(asset).approve(address(lendingPool), type(uint256).max);
66          lendingPool.deposit(asset, amount, msg.sender, referralCode);
67          for (uint256 i = 0; i < loopCount; i += 1) {
68              amount = amount.mul(borrowRatio).div(10 ** BORROW_RATIO_DECIMALS);
69              lendingPool.borrow(asset, amount, interestRateMode, referralCode, msg.sender
                    );
70              lendingPool.deposit(asset, amount, msg.sender, referralCode);
71          }
72      }
```

Listing 3.5: `Leverager::loop()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transferFrom()/approve()`. And there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Status The issue has been addressed in the following deployed contract: `Leverager.sol`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the UWU protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and price oracle adjustment). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function setAssetSources(address[] calldata assets, address[] calldata sources)
        external
        onlyOwner
    {
        _setAssetsSources(assets, sources);
    }

    function setFallbackOracle(address fallbackOracle) external onlyOwner {
        _setFallbackOracle(fallbackOracle);
    }
```

Listing 3.6: AaveOracle

Moreover, the LendingPoolAddressesProvider contract allows the privileged owner to configure protocol-wide contracts, including LENDING_POOL, LENDING_POOL_CONFIGURATOR, POOL_ADMIN, EMERGENCY_ADMIN, LENDING_POOL_COLLATERAL_MANAGER, PRICE_ORACLE, and LENDING_RATE_ORACLE. These contracts play a variety of duties and are also considered privileged.

```
    contract LendingPoolAddressesProvider is Ownable, ILendingPoolAddressesProvider {
        string private _marketId;
        mapping(bytes32 => address) private _addresses;

        bytes32 private constant LENDING_POOL = 'LENDING_POOL';
        bytes32 private constant LENDING_POOL_CONFIGURATOR = 'LENDING_POOL_CONFIGURATOR'
            ;
        bytes32 private constant POOL_ADMIN = 'POOL_ADMIN';
        bytes32 private constant EMERGENCY_ADMIN = 'EMERGENCY_ADMIN';
        bytes32 private constant LENDING_POOL_COLLATERAL_MANAGER = 'COLLATERAL_MANAGER';
        bytes32 private constant PRICE_ORACLE = 'PRICE_ORACLE';
        bytes32 private constant LENDING_RATE_ORACLE = 'LENDING_RATE_ORACLE';
        ...
    }
```

Listing 3.7: LendingPoolAddressesProvider

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Our analysis shows that the privileged `owner` is currently configured as `0xb8416eac2155e9636b5f728dd-29810bf7e3bc20d`, which is a proxy to a multi-sig (3/5) `GnosisSafe` account.

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 The issue has been confirmed by the team.

3.6 Revisited Logic of MultiFeeDistribution::withdraw()

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MultiFeeDistribution`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In the `UWU` protocol, the `MultiFeeDistribution` contract provides an incentive mechanism that rewards the staking of the supported `stakingToken` with certain reward tokens. In particular, one entry routine, i.e., `withdraw()`, is designed to claim the unlocked earned rewards. While examining its logic, we notice there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the `MultiFeeDistribution` contract. By design, the `mapping(address => LockedBalance[])private userEarnings` records the user's earned rewards. Inside the `withdraw()` routine, the claimable rewards are accumulated within the `for` loop (line 1178) while the locked rewards reach the unlocked time. After that, the total claimable rewards are transferred to the recipient (line 1186). However, we notice the `bal.earned` (saving the total earned rewards) is not updated accordingly, which makes it possible for the user to get more rewards. Given this, we suggest to improve the implementation as below: `bal.earned = bal.earned.sub(amount)` (line 1186).

```
1165     function withdraw() public {
1166         _updateReward(msg.sender);
1167         Balances storage bal = balances[msg.sender];
1168         uint earned = bal.earned;
```

```
        uint amount;
        if (earned > 0) {
            uint length = userEarnings[msg.sender].length;
            for (uint i = 0; i < length; i++) {
                uint earnedAmount = userEarnings[msg.sender][i].amount;
                if (earnedAmount == 0) continue;
                if (userEarnings[msg.sender][i].unlockTime > block.timestamp) {
                    break;
                }
                amount = amount.add(earnedAmount);
                delete userEarnings[msg.sender][i];
            }
            if (userEarnings[msg.sender].length == 0) {
                delete userEarnings[msg.sender];
            }
        }
        if (amount > 0) {
            rewardToken.safeTransfer(msg.sender, amount);
            emit Withdrawn(msg.sender, amount);
        }
    }
```

Listing 3.8: `MultiFeeDistribution::withdraw()`

Recommendation Revisit the implementation of the above `withdraw()` routine.

Status The issue has been addressed in the following deployed contract: `MultiFeeDistributionV2.sol`.

PeckShield

4 Conclusion

In this audit, we have analyzed the design and implementation of the `UWU` protocol, which is a decentralized non-custodial liquidity markets protocol that is developed on top of one of the largest DeFi protocols, i.e., `AAVE`. The `UWU` protocol allows users to participate as depositors or borrowers. Depositors provide liquidity to the market to earn passive income, while borrowers are able to borrow in an over-collateralized (perpetually) or under-collateralized (one-block liquidity) fashion. 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.

PeckShield

References

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

[2] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

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

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

[9] Volt Protocol team. Market Manipulation Risk. https://github.com/volt-protocol/volt-protocol-core/blob/develop/audits/venue-audits/compound.md.