SMART CONTRACT AUDIT REPORT

for

Illuvium Protocol (Staking Contracts V2)

Prepared By: Patrick Lou

PeckShield

March 19, 2022

Document Properties

Client	Illuvium
Title	Smart Contract Audit Report
Target	Illuvium
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 19, 2022	Xuxian Jiang	Final Release
1.0-rc	February 26, 2022	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `Illuvium` protocol's `staking contracts V2`, 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 Illuvium

`Illuvium` is an open-world `RPG` adventure game built on the `Ethereum` blockchain. The game has been designed with journeys across a vast and varied landscape on the player quest to hunt and capture deity-like creatures called `Illuvials`, as well as discover the cause of the cataclysm that shattered this land. The audited `smart contracts (V2)` extend and refactor much of the functionality of the existing system for improved features as well as gas efficiency. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Illuvium

Item	Description
Name	Illuvium
Website	https://www.illuvium.io/
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	March 19, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/IlluviumGame/staking-contracts-v2.git (8c9859c)

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

- https://github.com/IlluviumGame/staking-contracts-v2.git (a4ac29e)

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 accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

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) [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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 design and implementation of the `Illuvium` protocol's `staking contracts V2`, 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	3	■ ■ ■
Low	2	■ ■
Informational	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 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 3 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Timely Pool Update Upon Weight Change in CorePool/FlashPool	Business Logic	Fixed
PVE-002	Medium	Proper Migration in CorePool::moveFundsFromWallet()	Business Logic	Fixed
PVE-003	Low	Proper Stake Weight Calculation in CorePool::stake()	Coding Practices	Fixed
PVE-004	Medium	Just-In-Time Pair Pool Balance For Extra ILV Rewards	Time And State	Mitigated
PVE-005	Informational	Removal of Redundant State/Code	Coding Practices	Fixed
PVE-006	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 contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Timely Pool Update Upon Weight Change in CorePool/FlashPool

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: CorePool, FlashPool
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

The staking contracts in Illuvium behavior as the staking pools that allow users to stake specified value of tokens for the intended value of time and get in return pending yield rewards (if any). While reviewing the staking logic, we notice the current staking pools support the runtime reconfiguration of pool weights and the update logic to the pool weights needs to be improved.

To elaborate, we show below the related setWeight() function from the FlashPool contract. As the name indicates, this function adjusts or modifies the pool weight. However, when the pool weight is adjusted, there is a need to ensure the accounting of related staking rewards or revenue is timely updated before applying the new pool weight. Unfortunately, the current implementation does not timely update the accounting upon the pool weight update.

```
    function setWeight(uint32 _weight) external virtual {
        bytes4 fnSelector = this.setWeight.selector;
        // verify function is executed by the factory
        fnSelector.verifyState(msg.sender == address(_factory), 0);

        // set the new weight value
        weight = _weight;

        // emit an event logging old and new weight values
        emit LogSetWeight(msg.sender, weight, _weight);
```

```
    }
```

Listing 3.1: `FlashPool::setWeight()`

Note that two contracts `CorePool` and `FlashPool` share the same issue.

Recommendation Timely update the staking rewards/revenue accounting when there is a need to update the pool weight.

Status This issue has been fixed in the following commit: `ea01e91`.

3.2 Proper Migration in CorePool::moveFundsFromWallet()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CorePool`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

The staking contracts support a neat feature in allowing users to transfer their total positions to new addresses. This is useful in situations such as when a personal private key is leaked or anything that could motivate a user to move into a new address. While analyzing the migration logic, we notice the current implementation needs to be improved.

In particular, we show below the related `moveFundsFromWallet()` function that is tasked to support the migration. While the current implementation properly validates the migration conditions and transfer associated rewards, we notice the logic simply removes the `stakes` from the migrating user (line 582). In other words, the previous stakes that also need to migrated are simply removed!

```
533     function moveFundsFromWallet(address _to) public virtual {
534         // checks if the contract is in a paused state
535         _requireNotPaused();
536         // gets storage pointer to msg.sender user struct
537         User storage previousUser = users[msg.sender];
538         // gets storage pointer to desired address user struct
539         User storage newUser = users[_to];
540         // uses v1 weight values for rewards calculations
541         uint256 v1WeightToAdd = _useV1Weight(msg.sender);
542         // We process update global and user's rewards
543         // before moving the user funds to a new wallet.
544         // This way we can ensure that all v1 ids weight have been used before the v2
545         // stakes to a new address.
546         _updateReward(msg.sender, v1WeightToAdd);
547
```

```
        // we're using selector to simplify input and state validation
        bytes4 fnSelector = this.moveFundsFromWallet.selector;
        // validate input is set
        fnSelector.verifyNonZeroInput(uint160(_to), 0);
        // verify new user records are empty
        fnSelector.verifyState(
            newUser.totalWeight == 0 &&
                newUser.v1IdsLength == 0 &&
                newUser.stakes.length == 0 &&
                newUser.yieldRewardsPerWeightPaid == 0 &&
                newUser.vaultRewardsPerWeightPaid == 0,
            0
        );
        // saves previous user total weight
        uint248 previousTotalWeight = previousUser.totalWeight;
        // saves previous user pending yield
        uint128 previousYield = previousUser.pendingYield;
        // saves previous user pending rev dis
        uint128 previousRevDis = previousUser.pendingRevDis;

        // It's expected to have all previous user values
        // migrated to the new user address (_to).
        // We recalculate yield and vault rewards values
        // to make sure new user pending yield and pending rev dis to be stored
        // at newUser.pendingYield and newUser.pendingRevDis is 0, since we just
            processed
        // all pending rewards calling _updateReward.
        newUser.totalWeight = previousTotalWeight;
        newUser.pendingYield = previousYield;
        newUser.pendingRevDis = previousRevDis;
        newUser.yieldRewardsPerWeightPaid = yieldRewardsPerWeight;
        newUser.vaultRewardsPerWeightPaid = vaultRewardsPerWeight;
        delete previousUser.totalWeight;
        delete previousUser.pendingYield;
        delete previousUser.pendingRevDis;
        delete previousUser.stakes;

        // emits an event
        emit LogMoveFundsFromWallet(
            msg.sender,
            _to,
            previousTotalWeight,
            newUser.totalWeight,
            previousYield,
            newUser.pendingYield,
            previousRevDis,
            newUser.pendingRevDis
        );
    }
```

Listing 3.2: CorePool::moveFundsFromWallet()

Recommendation Revise the above `moveFundsFromWallet()` function to properly migrate the stakes as well.

Status This issue has been fixed in the following commit: `ea01e91`.

3.3 Proper Stake Weight Calculation in CorePool::stake()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CorePool`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned earlier, the staking contracts have the standard functionality that allows users to dynamically stake (or unstake) assets into (or out of) the staking pools. While analyzing the staking logic, we notice the current implementation can be improved for proper stake weight calculation.

To elaborate, we show below the related `stake()` function. This function implements a rather straightforward logic in staking specified value of tokens for the specified value of time, and pays pending yield rewards if any. However, it comes to our attention that the stake weight calculation is computed as follows: `stakeWeight = (((lockUntil - _now256())* Stake.WEIGHT_MULTIPLIER) / Stake.MAX_STAKE_PERIOD + Stake.WEIGHT_MULTIPLIER)* _value` (lines 496-498). The proper calculation should be revised as `stakeWeight = (((lockUntil - _now256())* Stake.WEIGHT_MULTIPLIER)/ Stake .MAX_STAKE_PERIOD + Stake.BASE_WEIGHT)* _value`.

```
476     function stake(uint256 _value, uint64 _lockDuration) external virtual nonReentrant {
477         // checks if the contract is in a paused state
478         _requireNotPaused();
479         // we're using selector to simplify input and state validation
480         bytes4 fnSelector = this.stake.selector;
481         // validate the inputs
482         fnSelector.verifyNonZeroInput(uint160(msg.sender), 0);
483         fnSelector.verifyNonZeroInput(_value, 1);
484         fnSelector.verifyInput(_lockDuration >= Stake.MIN_STAKE_PERIOD && _lockDuration
                <= Stake.MAX_STAKE_PERIOD, 2);

486         // get a link to user data struct, we will write to it later
487         User storage user = users[msg.sender];
488         // uses v1 weight values for rewards calculations
489         uint256 v1WeightToAdd = _useV1Weight(msg.sender);
490         // update user state
491         _updateReward(msg.sender, v1WeightToAdd);
```

```
        // calculates until when a stake is going to be locked
        uint64 lockUntil = (_now256()).toUint64() + _lockDuration;
        // stake weight formula rewards for locking
        uint256 stakeWeight = (((lockUntil - _now256()) * Stake.WEIGHT_MULTIPLIER) /
            Stake.MAX_STAKE_PERIOD +
            Stake.WEIGHT_MULTIPLIER) * _value;
        // makes sure stakeWeight is valid
        assert(stakeWeight > 0);
        // create and save the stake (append it to stakes array)
        Stake.Data memory userStake = Stake.Data({
            value: (_value).toUint120(),
            lockedFrom: (_now256()).toUint64(),
            lockedUntil: lockUntil,
            isYield: false
        });
        // pushes new stake to 'stakes' array
        user.stakes.push(userStake);
        // update user weight
        user.totalWeight += (stakeWeight).toUint248();
        // update global weight value and global pool token count
        globalWeight += stakeWeight;
        poolTokenReserve += _value;

        // transfer '_value'
        IERC20Upgradeable(poolToken).safeTransferFrom(address(msg.sender), address(this)
            , _value);

        // emit an event
        emit LogStake(msg.sender, msg.sender, (user.stakes.length - 1), _value,
            lockUntil);
    }
```

Listing 3.3: `CorePool::stake()`

It should be mentioned that the constant `Stake.WEIGHT_MULTIPLIER` is currently equal to `Stake.BASE_WEIGHT`. However, it is semantically incorrect to use `Stake.WEIGHT_MULTIPLIER`.

Recommendation Improve the above `stake()` routine to properly compute the stake weight.

Status This issue has been fixed in the following commit: `ea01e91`.

3.4 Just-In-Time Pair Pool Balance For Extra ILV Rewards

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Vault`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [3]

Description

The `Illuvium` vault plays a critical role in the collecting and distributing rewards to staking pools. In particular, it is responsible to gather revenue from the protocol, swap to the protocol token (`ILV`) periodically, and distribute to core pool users from time to time. When analyzing the revenue distribution logic, we notice the current implementation may be improved.

To elaborate, we show below the related `sendILVRewards()` function. This function converts an entire contract's `ETH` balance into `ILV` via the `Sushiswap` DEX and sends the entire contract's `ILV` balance to current yield pools. It comes to our attention the distribution logic is computed according to the estimated pair pool reserve (line 233), which unfortunately may be abused to provide just-in-time balance to inflate and cause unfair revenue distribution!

```
195     function sendILVRewards(
196         uint256 _ethIn,
197         uint256 _ilvOut,
198         uint256 _deadline
199     ) external onlyOwner {
200         // we treat set ‘ilvOut‘ and ‘deadline‘ as a flag to execute ‘swapEthForIlv‘
201         // in the same time we won’t execute the swap if contract balance is zero
202         if (_ilvOut > 0 && _deadline > 0 && address(this).balance > 0) {
203             // exchange ETH on the contract’s balance into ILV via Sushi - delegate to ‘
                    swapEthForIlv‘
204             _swapETHForILV(_ethIn, _ilvOut, _deadline);
205         }
206
207         // reads core pools
208         (ICorePool ilvPool, ICorePool pairPool, ICorePool lockedPoolV1, ICorePool
                lockedPoolV2) = (
209             pools.ilvPool,
210             pools.pairPool,
211             pools.lockedPoolV1,
212             pools.lockedPoolV2
213         );
214
215         // read contract’s ILV balance
216         uint256 ilvBalance = _ilv.balanceOf(address(this));
217         // approve the entire ILV balance to be sent into the pool
218         if (_ilv.allowance(address(this), address(ilvPool)) < ilvBalance) {
```

```
            _ilv.approve(address(ilvPool), ilvBalance);
        }
        if (_ilv.allowance(address(this), address(pairPool)) < ilvBalance) {
            _ilv.approve(address(pairPool), ilvBalance);
        }
        if (_ilv.allowance(address(this), address(lockedPoolV1)) < ilvBalance) {
            _ilv.approve(address(lockedPoolV1), ilvBalance);
        }
        if (_ilv.allowance(address(this), address(lockedPoolV2)) < ilvBalance) {
            _ilv.approve(address(lockedPoolV2), ilvBalance);
        }

        // gets poolToken reserves in each pool
        uint256 reserve0 = ilvPool.getTotalReserves();
        uint256 reserve1 = estimatePairPoolReserve(address(pairPool));
        uint256 reserve2 = lockedPoolV1.poolTokenReserve();
        uint256 reserve3 = lockedPoolV2.poolTokenReserve();

        // ILV in ILV core pool + ILV in ILV/ETH core pool representation + ILV in
            locked pool
        uint256 totalReserve = reserve0 + reserve1 + reserve2 + reserve3;

        // amount of ILV to send to ILV core pool
        uint256 amountToSend0 = _getAmountToSend(ilvBalance, reserve0, totalReserve);
        // amount of ILV to send to ILV/ETH core pool
        uint256 amountToSend1 = _getAmountToSend(ilvBalance, reserve1, totalReserve);
        // amount of ILV to send to locked ILV pool V1
        uint256 amountToSend2 = _getAmountToSend(ilvBalance, reserve2, totalReserve);
        // amount of ILV to send to locked ILV pool V2
        uint256 amountToSend3 = _getAmountToSend(ilvBalance, reserve3, totalReserve);

        // makes sure we are sending a valid amount
        assert(amountToSend0 + amountToSend1 + amountToSend2 + amountToSend3 <=
            ilvBalance);

        // sends ILV to both core pools
        ilvPool.receiveVaultRewards(amountToSend0);
        pairPool.receiveVaultRewards(amountToSend1);
        lockedPoolV1.receiveVaultRewards(amountToSend2);
        lockedPoolV2.receiveVaultRewards(amountToSend3);

        // emit an event
        emit LogSendILVRewards(msg.sender, ilvBalance);
    }
```

Listing 3.4: `Vault::sendILVRewards()`

In particular, the estimate is performed by directly measure the holding amount of `ILV` of the LP token contract (line 284). In a flashbot-assisted MEV situation, it is possible to simply flash borrow `ILV` right before the measurement and immediately return the borrow afterward for manipulated revenue distribution.

```
    function estimatePairPoolReserve(address _pairPool) public view returns (uint256 ilvAmount) {
        // 1. Store the amount of LP tokens staked in the ILV/ETH pool
        //    and the LP token total supply (total amount of LP tokens in circulation).
        //    With these two values we will be able to estimate how much ILV each LP token
        //    is worth.
        uint256 lpAmount = ICorePool(_pairPool).getTotalReserves();
        uint256 lpTotal = IERC20Upgradeable(ICorePool(_pairPool).poolToken()).totalSupply();

        // 2. We check how much ILV the LP token contract holds, that way
        //    based on the total value of ILV tokens represented by the total
        //    supply of LP tokens, we are able to calculate through a simple rule
        //    of 3 how much ILV the amount of staked LP tokens represent.
        uint256 ilvTotal = _ilv.balanceOf(ICorePool(_pairPool).poolToken());
        // we store the result
        ilvAmount = (ilvTotal * lpAmount) / lpTotal;
    }
```

Listing 3.5: `Vault::estimatePairPoolReserve()`

Recommendation Revisit the revenue distribution mechanism defensively against the above MEV issue.

Status The issue has been mitigated by the use of flashbots by the team to prevent any type of frontrunning.

3.5 Removal of Redundant State/Code

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CorePool`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

The new staking contracts make good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, and `Initializable`, to facilitate its code implementation and organization. For example, the `CorePool` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `CorePool` contract, there is a public `stake()` function that is designed to allow users to stake specified value of tokens for the intended value of time and get in return pending yield rewards if any. However, it comes to our attention that it has a specific requirement in validating `uint160(msg.sender)!= 0` (line 482), which is always evaluated to be false. As a result, this specific validation can be safely removed.

```
476     function stake(uint256 _value, uint64 _lockDuration) external virtual nonReentrant {
477         // checks if the contract is in a paused state
478         _requireNotPaused();
479         // we're using selector to simplify input and state validation
480         bytes4 fnSelector = this.stake.selector;
481         // validate the inputs
482         fnSelector.verifyNonZeroInput(uint160(msg.sender), 0);
483         fnSelector.verifyNonZeroInput(_value, 1);
484         fnSelector.verifyInput(_lockDuration >= Stake.MIN_STAKE_PERIOD && _lockDuration
                <= Stake.MAX_STAKE_PERIOD, 2);
485         ...
486     }
```

Listing 3.6: `CorePool::stake()`

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status This issue has been fixed in the following commit: `ea01e91`.

3.6 Trust Issue Of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the new staking contracts of `Illuvium`, there exist certain privileged accounts that play critical roles in governing and regulating the system-wide operations. It also has the privilege to regulate or govern the flow of assets within the protocol. In the following, we show representative privileged operations in the protocol.

```
    function registerPool(address pool) public virtual onlyOwner {
        // read pool information from the pool smart contract
        // via the pool interface (ICorePool)
        address poolToken = ICorePool(pool).poolToken();
        bool isFlashPool = ICorePool(pool).isFlashPool();
        uint32 weight = ICorePool(pool).weight();

```

```
        // create pool structure, register it within the factory
        pools[poolToken] = pool;
        poolExists[pool] = true;
        // update total pool weight of the factory
        totalWeight += weight;

        // emit an event
        emit LogRegisterPool(msg.sender, poolToken, address(pool), weight, isFlashPool);
    }
```

Listing 3.7: `PoolFactory::registerPool()`

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. We point out that a compromised `owner` account would allow the attacker to utterly change the protocol configuration, which directly undermines the assumption of the `Illuvium` protocol.

Recommendation Make the list of extra privileges granted to `owner` explicit to `Illuvium` users.

Status This issue has been mitigated with the use of a multisig account as the owner instead of EOA.

4 Conclusion

In this audit, we have analyzed the design and implementation of the new staking contracts in the Illuvium protocol, which is an open-world RPG adventure game built on the Ethereum blockchain. The current code base is well structured and neatly organized. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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