Crowdswap Lock Staking

Smart Contract Security Audit

Prepared by ShellBoxes

Jan 10th, 2024 - Jan 15th, 2024

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Crowdswap
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
https://github.com/CrowdSwap/Opportunities/tree/BLOC-2596	a3197a2a13fb5333f787c41162d8d257b5a1b8d1

Re-Audit

Repository	Commit Hash
https://github.com/CrowdSwap/Opportunities/tree/BLOC-2596	23e15ce0dbaf0a78d6df8b483df54eeb1887ed02

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Crowdswap engaged ShellBoxes to conduct a security assessment on the Crowdswap Lock Staking beginning on Jan 10th, 2024 and ending Jan 15th, 2024. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Crowdswap

CrowdSwap is a cross-chain opportunity optimization and automation platform. It aims to reach mass adoption in crypto for every human being and overcome actual problems that reside from a fast-growing business space like DeFi.

Issuer	Crowdswap
Website	`https://crowdswap.org`
Type	Solidity Smart Contracts
Whitepaper	Crowdswap Whitepaper
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

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

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the Crowdswap Lock Staking implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include 1 critical-severity, 2 high-severity, 4 medium-severity, 1 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Changing APR for Existing Staking Plans Can Affect Users' Rewards	CRITICAL	Fixed
SHB.2. Multiple Front-Running Vulnerabilities in Transactions	HIGH	Fixed
SHB.3. Potential Locking of Funds due to whenNotPaused Modifier in withdraw Function	HIGH	Fixed
SHB.4. Inadequate Parameter Validation Across Multiple Functions	MEDIUM	Fixed
SHB.5. Inconsistency in Plan Uniqueness Validation for updatePlan Function	MEDIUM	Fixed
SHB.6. Potential Precision Loss in Fee and Reward Calculations	MEDIUM	Fixed

SHB.7. Potential Denial of Service (DoS) Risk Due to Iterative Searches	MEDIUM	Fixed
SHB.8. Mismatch Between The Code and Documentation	LOW	Fixed

3 Finding Details

SHB.1 Changing APR for Existing Staking Plans Can Affect Users' Rewards

- Severity : CRITICAL
- Status : Fixed
- Likelihood : 3
- Impact : 3

Description:

The updatePlan function allows the contract owner to update the attributes of a staking plan, including the APR (Annual Percentage Rate). If the owner decides to change the APR for an existing staking plan, it can affect users who have already staked tokens based on the previous APR. Their rewards will be calculated based on the new APR, which may not align with their original expectations.

Files Affected:

SHB.1.1: LockableStakingRewards.sol

```
        plans[_planId] = Plan(
            _planId,
            _newDuration,
            _newApr,
            _newDefaultApr,
            _newActive,
            true
        );
```

Recommendation:

When the owner updates the APR for an existing staking plan, consider implementing a mechanism to ensure that users who have already staked tokens under that plan continue to earn rewards based on the original APR they agreed to when staking. One approach is to introduce a snapshot mechanism that records the APR at the time of the user's stake. This snapshot can be used to calculate rewards for existing stakers, even if the plan's APR is updated later.

Updates

The team has mitigated the risk by removing the updatePlan function.

SHB.2 Multiple Front-Running Vulnerabilities in Transactions

- Severity : HIGH
- Status : Fixed
- Likelihood : 2
- Impact : 3

Description:

The withdraw, extend, and stake functions in the LockableStakingRewards contract are susceptible to front-running attacks. These vulnerabilities arise due to the mutable nature of key parameters (feeInfo.unstakeFee, feeInfo.stakeFee, and plan.duration) that can be changed by the contract owner. In withdraw and extend, the issue lies with the potential alteration of fee amounts between the initiation and execution of a transaction. In stake, the vulnerability is related to the potential change in plan duration.

- In withdraw and extend Functions: A malicious actor or the contract owner can observe a pending transaction and increase the fee values (unstakeFee or stakeFee), leading to users paying higher fees than expected.
- In stake Function: The plan duration can be modified by the owner after a user initiates a staking transaction, but before it is executed, potentially locking user funds for longer than intended.

Files Affected:

SHB.2.1: LockableStakingRewards.sol

```
/**
 * @param _planId The id of a specific plan
 * @param _amount The amount to stake
 */
function stake(
    uint256 _planId,
    uint256 _amount
) external nonReentrant whenNotPaused validPlanId(_planId) {
    require(
        _amount > 0,
        "LockableStakingRewards: Staked amount must be greater than
            ↪ 0."
    );

    _transferTokenFromTo(
        stakingToken,
        msg.sender,
        payable(address(this)),
        _amount
    );

    //Decrease fee
    (_amount, ) = _deductFee(feeInfo.stakeFee, stakingToken, _amount)
        ↪ ;
```

SHB.2.2: LockableStakingRewards.sol

```
/**
 * @notice if the user wants to extend in specific plan
 * @param _stakeId The id of a specific stake
 */
function extend(
```

```
        uint256 _stakeId
    )
        external
        nonReentrant
        whenNotPaused
        validStakeId(_stakeId)
        validUser(msg.sender)
    {
        Stake storage _stakeToRestake = _getUserStakeByStakeId(
            msg.sender,
            _stakeId
        );

        Plan memory _plan = plans[_stakeToRestake.planId];
```

SHB.2.3: LockableStakingRewards.sol

```
    /**
     * @notice if the amount be equal to origin staked amount, or the
       ↪ _max is true, reward and origin staked amount will withdraw
     * @param _stakeId , an uniq id for each stake
     * @param _amount to withdraw
     * @param _max A boolean to withdraw the max amount
     */
    function withdraw(
        uint256 _stakeId,
        uint256 _amount,
        bool _max
    )
        external
        nonReentrant
        whenNotPaused
        validStakeId(_stakeId)
        validUser(msg.sender)
    {
```

```
        /**
         * When _max is true, the _amount is not important.
         * It will be calculated later and replaced by the calculated max amount
         */
        require(
            _max  _amount > 0,
            "LockableStakingRewards: withdraw amount must be greater than 0."
        );
```

Recommendation:

- For fee-related front-running: Implement a mechanism to lock the fee amount for a certain period or a specific number of blocks once a transaction is initiated. This ensures that the fee at the time of transaction initiation remains unchanged until execution.
- For plan duration front-running: Consider locking the plan's parameters (like duration) once a user stakes in it, preventing any changes to these parameters that would affect existing stakes.

In addition to locking key parameters at the start of a transaction, consider allowing users to specify expected values for critical parameters (like fees and plan durations) as part of the function call. The contract should then compare these user-provided values with the actual values in the contract at the time of execution. If there is a discrepancy between the user's expectations and the contract's current state, the transaction should revert. This approach empowers users to set their terms based on the information available at the time of transaction initiation and protects them from changes made in the interim.

SHB.2.4: LockableStakingRewards.sol

```
// Modified withdraw function
function withdraw(uint256 \_stakeId, uint256 \_amount, bool \_max, uint256 expectedUnstakeFee, uint256 expectedStakeFee) external {
require(feeInfo.unstakeFee == expectedUnstakeFee, "LockableStakingRewards: Unstake fee changed");
```

```
require(feeInfo.stakeFee == expectedStakeFee, "LockableStakingRewards:
    ↪ Stake fee changed");
...
(\_amount, ) = \_deductFee(expectedUnstakeFee, \_amount);
...
}
// Modified extend function
function extend(uint256 \_stakeId, uint256 expectedUnstakeFee, uint256
    ↪ expectedStakeFee) external {
require(feeInfo.unstakeFee == expectedUnstakeFee, "
    ↪ LockableStakingRewards: Unstake fee changed");
require(feeInfo.stakeFee == expectedStakeFee, "LockableStakingRewards:
    ↪ Stake fee changed");
...
(\_maxWithdrawAmount, ) = \_deductFee( \_feeInfo.unstakeFee + \_feeInfo.
    ↪ stakeFee, stakingToken, \_maxWithdrawAmount );
...
}
// Modified \_createStake function
function stake(..., uint128 expectedPlanDuration) internal {
require(plans[\_planId].duration == expectedPlanDuration, "
    ↪ LockableStakingRewards: Plan duration changed");
...
Stake memory \_newStake = Stake({
...
endTime: block.timestamp + expectedPlanDuration,
...
});
...
}
```

Updates

The team has fixed the issue by removing the updatePlan function and added the whenPaused modifier in the setFee function.

SHB.3 Potential Locking of Funds due to whenNotPaused Modifier in withdraw Function

- Severity : HIGH
- Status : Fixed
- Likelihood : 2
- Impact : 3

Description:

The withdraw function in the LockableStakingRewards contract is guarded by the whenNotPaused modifier. This design means that if the contract is paused, users will not be able to withdraw their staked funds. While pausing the contract is a useful feature in case of emergencies or detected vulnerabilities, it also poses a risk of locking user funds indefinitely if the contract remains paused for an extended period or permanently. This could lead to a loss of trust and potential financial losses for users.

Files Affected:

SHB.3.1: LockableStakingRewards.sol

```
    function withdraw(
        uint256 _stakeId,
        uint256 _amount,
        bool _max
    )
        external
        nonReentrant
        whenNotPaused
```

Recommendation:

- Consider implementing an emergency withdrawal function that is not affected by the whenNotPaused modifier. This function should allow users to withdraw their funds even when the contract is paused. However, it is crucial to ensure that this emergency function does not compromise the security of the contract.
- Alternatively, establish clear governance or administrative procedures for unpausing the contract and communicate these to the users. This ensures that users are aware of the conditions under which their funds can be locked and the measures in place for unlocking them.

Updates

The team has fixed the issue by removing the whenNotPaused modifier in the withdraw function.

SHB.4 Inadequate Parameter Validation Across Multiple Functions

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 2
- Impact : 2

Description:

Several functions in the LockableStakingRewards contract lack adequate validation for their input parameters. This includes the initialize, createPlan, updatePlan, and setFee functions. Specifically, the APR values in createPlan and updatePlan are not validated against realistic upper bounds (such as 100%), additionally the setFee and initialize functions lacks validation for its fee parameters and staking token address.

Files Affected:

SHB.4.1: LockableStakingRewards.sol

```
    function initialize(
        address _stakingToken,
        FeeInfo memory _feeInfo
    ) public initializer {
        require(
            _feeInfo.feeTo != address(0),
            "LockableStakingRewards: feeTo address is not valid"
        );
```

SHB.4.2: LockableStakingRewards.sol

```
        uint16 _apr,
        uint16 _defaultApr
```

SHB.4.3: LockableStakingRewards.sol

```
        uint16 _newApr,
        uint16 _newDefaultApr,
```

SHB.4.4: LockableStakingRewards.sol

```
        feeInfo = _feeInfo;
```

Recommendation:

- For APR-related parameters in createPlan and updatePlan, add validation to ensure they do not exceed realistic limits (e.g., 100%).
- In the setFee function, validate all fields of _feeInfo to ensure they meet the contract's expectations (e.g., fee percentages within acceptable ranges. Similarly, in the initialize function, perform the same validation for _feeInfo and additionally, verify that _stakingToken is not a zero address.

Updates

The team has fixed the issue by adding the _requiredValidFee, _requiredValidApr and _requiredValidAddress functions.

SHB.5 Inconsistency in Plan Uniqueness Validation for updatePlan Function

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 2
- Impact : 2

Description:

The updatePlan function in the LockableStakingRewards contract allows updating the attributes of an existing plan. However, it does not check if the new attributes (specifically the combination of _newDuration, _newApr, and _newDefaultApr) would result in a plan that duplicates another existing plan. This omission could lead to multiple plans with identical parameters, which seems to be against the intended design, as the createPlan function uses _planExists to ensure uniqueness of plans.

Files Affected:

SHB.5.1: LockableStakingRewards.sol

```
    function updatePlan(
        uint256 _planId,
        uint128 _newDuration,
        uint16 _newApr,
        uint16 _newDefaultApr,
        bool _newActive
    ) external onlyOwner validPlanId(_planId) {
        require(
            plans[_planId].exists,
            "LockableStakingRewards: Plan does not exist."
        );
        require(
            _newDuration > 0,
            "LockableStakingRewards: Invalid new duration."
```

```
        );
        require(_newApr > 0, "LockableStakingRewards: Invalid new apr.");
        require(
            _newDefaultApr >= 0,
            "LockableStakingRewards: Invalid default new apr."
        );

        // Update the plan attributes
        plans[_planId] = Plan(
            _planId,
            _newDuration,
            _newApr,
            _newDefaultApr,
            _newActive,
            true
        );
```

Recommendation:

Introduce a validation step in updatePlan to check if a plan with the new parameters already exists, similar to the check in createPlan. This could be done by calling a modified version of the _planExists function.

Updates

The issue was mitigated by removing the updatePlan function.

SHB.6 Potential Precision Loss in Fee and Reward Calculations

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 2
- Impact : 2

Description:

In the contract, there are instances where there is a potential for precision loss in fee and reward calculations due to the use of integer division or a fixed divisor. This precision loss can affect the accuracy of fee calculations and reward distribution, especially when dealing with small values. It's important to ensure that financial calculations maintain precision to provide fair and accurate results.

Exploit Scenario:

- The attacker prepares a transaction and specifies a fee calculation using the _calculateFee function with the following values:
- _amount: 100 tokens
- _percentage: 10^{17} (0.1%)
- The attacker sends the transaction to the contract.
- Inside the _calculateFee function, the fee is calculated as:

$$Fee = (100 * 10^{17})/10^{20} = 0$$

- The contract charges a fee of 0 tokens based on the calculation, which is lower than the expected fee for 0.1% of 100 tokens (expected fee: 0.1 tokens).
- The attacker successfully pays a lower fee than intended, exploiting the precision loss in the _calculateFee function.

Files Affected:

SHB.6.1: LockableStakingRewards.sol

```
    function _calculateFee(
        uint256 _amount,
        uint256 _percentage
    ) internal pure returns (uint256) {
        return (_amount * _percentage) / 1e20;
    }
```

SHB.6.2: LockableStakingRewards.sol

```
    function _getReward(
        uint256 _amount,
        uint256 _apr,
        uint256 _duration
    ) internal pure returns (uint256 _reward) {
        _reward = (_amount * _apr * _duration) / (365 days * 1e4);
    }
```

Recommendation:

To avoid precision loss in fee and reward calculations, consider using fixed-point arithmetic with a higher precision and appropriate factors to maintain precision.

Updates

The team has fixed the issue by verifying the fees if equal to 0 the transaction will be reverted.

SHB.7 Potential Denial of Service (DoS) Risk Due to Iterative Searches

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 1
- Impact : 3

Description:

The contract uses iterative searches in multiple functions, such as _findStakeIndex, _planExists. As the size of certain data structures (e.g., stake list or plans) grows, these iterative searches can lead to high gas costs and make transactions expensive or even infeasible to execute within the gas limit. In extreme cases, this could be exploited in a DoS attack, where an attacker creates a large number of elements (stakes or plans) to make these functions costly or impractical to execute.

Files Affected:

SHB.7.1: LockableStakingRewards.sol

```
        for (uint256 i = 0; i < planCounter; ++i) {
            Plan storage currentPlan = plans[i];

            if (
                currentPlan.exists &&
                currentPlan.duration == _duration &&
                currentPlan.apr == _apr &&
                currentPlan.defaultApr == _defaultApr
            ) {
                return true; // A plan with these attributes already
                    ↪ exists
            }
        }
```

SHB.7.2: LockableStakingRewards.sol

```
        for (uint256 i = 0; i < _stakeList.length; ++i) {
            if (_stakeList[i].id == _stakeId) {
                return i;
            }
        }
```

Recommendation:

- Consider using mappings to allow for constant-time lookups instead of iterative searches. For example, you can use a mapping from stake ID to stake index or directly to the stake structure, if feasible. Similarly, you can use mappings for plans or other elements where applicable.

- Additionally, consider setting limits or constraints on the maximum number of elements (e.g., stakes or plans) that can be created to mitigate potential DoS attacks.

Updates

The team has fixed the issue by changing the structure of plans to a mapping, thus removing the for loop, the same also for the _findStakeIndex function.

SHB.8 Mismatch Between The Code and Documentation

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 1

Description:

The totalInvestCount attribute in the InvestmentInfo struct serves as a counter for the number of stakes in each plan. It is incremented with each new stake in the _createStake function. However, there is a discrepancy in the documentation where the InvestmentInfo struct's NatSpec comment describes totalInvestCount as the number of stakers in each plan. This inconsistency between the documentation and the actual implementation can lead to confusion and potential misinterpretation.

Files Affected:

SHB.8.1: LockableStakingRewards.sol

```
/**
 * @dev a struct containing the investmanet details of each plan
 * @member totalInvestAmount , is the total amount of staked in each
    ↪ plan
 * @member totalInvestCount , is the number of stakers in each plan
 */
struct InvestmentInfo {
```

```
        uint256 totalInvestAmount;
        uint256 totalInvestCount;
    }
```

SHB.8.2: LockableStakingRewards.sol

```
        planInvestmentInfo[_plan.id].totalInvestAmount += _stakedAmount;
        ++planInvestmentInfo[_plan.id].totalInvestCount;
```

Recommendation:

Consider updating the NatSpec comment for the totalInvestCount attribute in the InvestmentInfo struct to accurately reflect its purpose as the total number of stakes in each plan. This modification will align the documentation with the implementation, providing clarity to developers and users and ensuring accurate understanding of the code's functionality.

Updates

The team has fixed the issue by changing the Natspec to the correct meaning of totalInvestCount .

4 Best Practices

BP.1 Redundant Variable in _createStake Function

Description:

In the _createStake function, there is an additional variable _stakedAmount created to store the staked amount, which is then immediately assigned the value of _amount, furthermore _stakedAmount is inserted as input of the function and returned also without any changes. This additional variable is not necessary and can be eliminated for code simplification. Remove the unnecessary variable _stakedAmount and directly assign the _amount parameter.

Files Affected:

BP.1.1: LockableStakingRewards.sol

```
    function _createStake(
        Plan memory _plan,
        address user,
        uint256 _amount
    ) internal returns (uint256 _stakeId, uint256 _stakedAmount) {
        // Check if the address has staked before
        if (!hasStaked[user]) {
            hasStaked[user] = true;
            stakerList.push(user); // Add the address to the array
        }

        _stakeId = stakeCounter++;
        _stakedAmount = _amount;
```

Status - Acknowledged

BP.2 Redundant ID Field in Plan Struct

Description:

The Plan struct in the LockableStakingRewards contract includes an id field. However, this id seems redundant since the Plan struct instances are already being mapped by their IDs through the mapping(uint256 => Plan) public plans;. This redundancy leads to unnecessary storage usage and potential confusion. The ID of each plan can be efficiently inferred from the key used in the mapping, making the explicit id field in the struct unnecessary. Consider removing the id field from the Plan struct to reduce storage costs and simplify the data model. Ensure that all parts of the contract that reference this id are refactored accordingly. This would involve iterating through the keys of the plans mapping wherever necessary to access or display plan IDs.

Files Affected:

BP.2.1: LockableStakingRewards.sol

```
    struct Plan {
        uint256 id;
        uint128 duration;
        uint16 apr;
        uint16 defaultApr;
        bool active;
        bool exists;
    }
```

Status - Acknowledged

BP.3 Redundancy and Potential Inconsistency in InvestmentInfo Struct

Description:

The InvestmentInfo struct is used to store totalInvestAmount and totalInvestCount for each plan. However, these values could potentially be derived from other data structures within the contract, such as iterating over user stakes. Storing them separately increases the risk of data inconsistency, especially if updates to these values are not handled atomically with corresponding stake updates. Additionally, incorporating these fields directly into the Plan struct could streamline data handling and reduce the need for separate mappings.

- Consider integrating totalInvestAmount and totalInvestCount directly into the Plan struct. This approach could simplify the contract's data model and reduce the likelihood of inconsistencies.
- Ensure that these values are updated consistently whenever stakes are created, modified, or withdrawn. Alternatively, if these values are kept separate for read-optimization, ensure strict consistency in updating these values alongside stake modifications.

Files Affected:

BP.3.1: LockableStakingRewards.sol

```
    struct InvestmentInfo {
        uint256 totalInvestAmount;
        uint256 totalInvestCount;
    }
```

Status - Acknowledged

BP.4 Redundancy of hasStaked Mapping

Description:

The hasStaked mapping in the contract is used to keep track of whether an address has ever staked. However, this information is redundant, since the same can be inferred from the userStakes mapping. By checking if the length of the array in userStakes[userAddress] is greater than zero, one can determine if the user has staked before. The hasStaked mapping thus introduces unnecessary state storage, leading to slightly increased gas costs for stakers and contract deployment. Consider removing the hasStaked mapping. Instead, use the userStakes mapping to check if a user has staked before by examining the length of the stake array for that user. Ensure to update all contract functions that currently rely on hasStaked to use this new method of checking.

BP.4.1: LockableStakingRewards.sol

```
// Instead of using hasStaked[userAddress]
if (userStakes[userAddress].length > 0) {
// User has staked before
}
```

Files Affected:

BP.4.2: LockableStakingRewards.sol

```
mapping(address => bool) public hasStaked;
```

Status - Acknowledged

BP.5 Redundancy of planCounter with Mapping-based Plan Storage

Description:

The planCounter variable is used to keep track of the number of plans and to generate unique IDs for new plans. However, this approach may be redundant if plans were implemented as an array instead of a mapping. An array naturally tracks the count and indexes, which can serve as plan IDs. This change could simplify the contract by eliminating the need for a separate counter variable and potentially streamline the process of adding, removing, or accessing plans. Consider changing the plans mapping to an array of Plan structs (Plan[] public plans;). This way, the length of the array (plans.length) would automatically serve as the count of plans, and the index in the array would act as the plan ID. Ensure to update all contract functions that interact with plans to accommodate this change.

BP.5.1: LockableStakingRewards.sol

```
// Using an array instead of a mapping
Plan[] public plans;
// Adding a new plan
plans.push(Plan({
duration: \_duration,
apr: \_apr,
defaultApr: \_defaultApr,
active: true,
exists: true
}));
// Accessing a plan by ID (index)
Plan memory selectedPlan = plans[planId];
```

Files Affected:

BP.5.2: LockableStakingRewards.sol

```
    uint256 public planCounter;
    mapping(uint256 => Plan) public plans;
```

Status - Acknowledged

BP.6 Inefficient Order of Operations and Potential Optimization in createPlan Function

Description:

In the createPlan function of the LockableStakingRewards contract, the order of operations can be optimized for gas efficiency. Currently, the function first increments planCounter and then checks if a similar plan already exists using _planExists. This approach can lead to unnecessary gas consumption if the plan already exists. Furthermore, there's a suggestion to optimize the uniqueness check of plans using hashing, but this requires careful consideration.

- Rearrange the order of operations to perform the _planExists check before incrementing the planCounter. This ensures that gas is not wasted on incrementing the counter for a plan that already exists.
- Consider replacing the use of integers as keys in the mapping with a hashed representation (e.g., using keccak3) for various attributes such as APR and duration. Utilize these hashed values as keys to access plans within the mapping.

Files Affected:

BP.6.1: LockableStakingRewards.sol

```
    function createPlan(
        uint128 _duration,
        uint16 _apr,
```

```
        uint16 _defaultApr
    ) external onlyOwner {
        require(_duration > 0, "LockableStakingRewards: Invalid duration
            ↪ .");
        require(_apr >= 0, "LockableStakingRewards: Invalid apr.");
        require(
            _defaultApr >= 0,
            "LockableStakingRewards: Invalid default apr."
        );

        // Using planCounter as the new planId
        uint256 _planId = planCounter++;

        require(
            !_planExists(_duration, _apr, _defaultApr),
            "LockableStakingRewards: Similar plan already exists."
        );
```

Status - Acknowledged

BP.7 Optimizing Plan Existence Check in updatePlan Function

Description:

In the updatePlan function of the LockableStakingRewards contract, the plan existence check is conducted by verifying plans[_planId].exists. An alternative approach could be to validate the existence of a plan by checking if a key attribute, such as apr, is different from an uninitialized state (e.g., apr != 0). This approach may streamline the validation process, especially if exists is only used for this purpose.

- Modify the validPlanId modifier to check if the apr of the plan is different from zero (or some other uninitialized state) instead of checking plans[_planId].exists.

- Ensure that this change accurately reflects plan existence and consider edge cases where `apr` might legitimately be zero.

Files Affected:

BP.7.1: LockableStakingRewards.sol

```
        require(
            plans[_planId].exists,
            "LockableStakingRewards: Plan does not exist."
```

BP.7.2: LockableStakingRewards.sol

```
        require(_plan.exists, "LockableStakingRewards: Plan does not exist.");
        require(_plan.active, "LockableStakingRewards: Plan does not active.");
```

BP.7.3: LockableStakingRewards.sol

```
    function _planExists(
        uint256 _duration,
        uint256 _apr,
        uint256 _defaultApr
    ) internal view returns (bool) {
        for (uint256 i = 0; i < planCounter; ++i) {
            Plan storage currentPlan = plans[i];

            if (
                currentPlan.exists &&
```

BP.7.4: LockableStakingRewards.sol

```
        require(_plan.exists, "LockableStakingRewards: Plan does not exist.");
```

Status - Acknowledged

BP.8 Redundant Token Argument in Functions

Description:

In the LockableStakingRewards contract, _transferTokenTo, _transferTokenFromTo and _deductFee functions are called with stakingToken as an argument. This is redundant because stakingToken is a fixed contract-wide state variable, and its value does not change across different function calls. Passing it as an argument in these functions is unnecessary and results in slightly higher gas costs due to the additional data handling. Simplifying these function calls by removing the redundant token parameter can improve gas efficiency and code clarity.

- Modify both _transferTokenFromTo, _transferTokenTo and _deductFee functions to use the stakingToken state variable directly. This eliminates the need to pass stakingToken as an argument.
- Update all instances where these functions are called to reflect the change in their signatures.

Files Affected:

BP.8.1: LockableStakingRewards.sol

```
    function _transferTokenFromTo(
        IERC20Upgradeable _token,
```

BP.8.2: LockableStakingRewards.sol

```
    function _transferTokenTo(
        IERC20Upgradeable _token,
```

BP.8.3: LockableStakingRewards.sol

```
    function _deductFee(
        uint256 _percentage,
        IERC20Upgradeable _token,
```

Status - Acknowledged

BP.9 Unnecessary _max Parameter in withdraw Function

Description:

The withdraw function in the LockableStakingRewards contract includes a boolean parameter _max to indicate whether the user wishes to withdraw the maximum possible amount. However, this functionality could be handled in the user interface or front-end application, allowing users to simply enter the maximum amount directly as the _amount parameter. The inclusion of the _max parameter in the contract adds unnecessary complexity and may slightly increase the gas cost due to the additional logic required to process this parameter.

- Consider removing the _max parameter from the withdraw function. Instead, allow users to specify the exact amount they wish to withdraw directly.
- The front-end application or user interface can calculate the maximum withdrawable amount and fill in this value for the user, simplifying the contract's logic and reducing gas costs.
- Ensure that the contract's documentation and user interface clearly communicate how users can perform a maximum withdrawal.

Files Affected:

BP.9.1: LockableStakingRewards.sol

```
    function withdraw(
        uint256 _stakeId,
        uint256 _amount,
        bool _max
```

Status - Acknowledged

BP.10 Redundant Check for Archived Stake in withdraw Function

Description:

The withdraw function in the LockableStakingRewards contract uses an archived flag to indicate whether a stake has been fully withdrawn. However, this flag may be redundant if the condition for a fully withdrawn stake (i.e., the user has unstaked their entire balance including rewards) can be inferred from other variables. The use of an additional flag for this purpose leads to extra gas consumption each time a stake's state is updated. Optimizing this process to infer the stake's completion from existing variables can save gas and streamline the contract's logic.

- Eliminate the archived flag and instead check if the user has fully withdrawn their stake. This can be done by comparing the paidAmount of the stake against the total staked amount plus rewards.
- Adjust the withdraw function to rely on this check, removing the need for the archived flag.

Files Affected:

BP.10.1: LockableStakingRewards.sol

```
        require(
            !_stakeToWithdraw.archived,
            "LockableStakingRewards: The stake has been archived"
        );
```

Status - Acknowledged

BP.11 Improving Clarity and Clean Code by Converting _planExists to a Modifier

Description:

The current implementation of _planExists as a function in the LockableStakingRewards contract, while functionally sound, can be enhanced in terms of code clarity and cleanliness by converting it into a modifier. Using a modifier for existence checks offers clearer intent, reduces repetition, and integrates the check more seamlessly into the function flow. This approach aligns with clean code principles, making the contract more maintainable and understandable, especially for other developers or auditors who may interact with the code in the future.

- Convert the _planExists function to a modifier, to be used in functions like createPlan and potentially others where similar checks are required.
- This modification should maintain the current logic but embed it directly into the function execution flow, providing a clearer indication of its purpose and usage.

Files Affected:

BP.11.1: LockableStakingRewards.sol

```
    function _planExists(
        uint256 _duration,
        uint256 _apr,
        uint256 _defaultApr
    ) internal view returns (bool) {
        for (uint256 i = 0; i < planCounter; ++i) {
            Plan storage currentPlan = plans[i];

            if (
                currentPlan.exists &&
                currentPlan.duration == _duration &&
```

```
                currentPlan.apr == _apr &&
                currentPlan.defaultApr == _defaultApr
            ) {
                return true; // A plan with these attributes already
                    ↪ exists
            }
        }

        return false; // No matching plan found
    }
```

Status - Acknowledged

BP.12 Introduce a Dedicated Deactivation Function for Plans

Description:

To enhance modularity and improve the clarity of plan management, it is recommended to create a separate function for deactivating plans. The existing updatePlan function, which currently takes a newActive parameter to modify the plan's activity status, can be optimized by introducing a new function specifically designed for deactivation. This dedicated deactivation function should only require the planId as a parameter, allowing the owner to deactivate a plan without the need to input all of its values.

Files Affected:

BP.12.1: LockableStakingRewards.sol

```
    /**
     * @notice To update an existing plan
     * @param _planId The id of specific plan
     * @param _newDuration, The duration of each staking plan (between
        ↪ startTime and endTime)
```

```
     * @param _newApr The apr of the staking plan, with 2 decimals, (1=0.01% )
     * @param _newDefaultApr The defaukt apr of each staking plan with 2 decimals, (1=0.01% ). After of duration, rewards will be calculate based on defaultApr
     */
    function updatePlan(
        uint256 _planId,
        uint128 _newDuration,
        uint16 _newApr,
        uint16 _newDefaultApr,
        bool _newActive
    ) external onlyOwner validPlanId(_planId) {
```

Status - Acknowledged

5 Tests

Results:

→ createPlan

✓ should allow the owner to create a new staking plan

✓ should not allow duplicate plan IDs

→ stake

✓ should allow a user to stake tokens

✓ should stake

✓ should allow a user to stake tokens

✓ should not allow staking for an invalid plan

✓ should allow multiple users to stake

✓ should retrieve and verify the details of a staked amount

✓ should add address to addresses array when staking

→ withdraw

✓ should allow a user to withdraw totally

✓ should calculate reward correctly for valid stake

✓ after withdraw couldn't staked by checking the archived flag

→ reStake

✓ should allow a user to reStake

→ UserDetails

✓ should return staking records for a user

✓ should return correct total staked amount for a user

6 Conclusion

In this audit, we examined the design and implementation of Crowdswap Lock Staking contract and discovered several issues of varying severity. Crowdswap team addressed all the issues raised in the initial report and implemented the necessary fixes.

However Shellboxes' auditors advised Crowdswap Team to maintain a high level of vigilance and participate in bounty programs in order to avoid any future complications.

7 Scope Files

7.1 Audit

Files	MD5 Hash
lock-stake/LockableStakingRewards.sol	35b4f6ec312bc9a875eab8935574555a

7.2 Re-Audit

Files	MD5 Hash
lock-stake/LockableStakingRewards.sol	2e7a7da94800521e8920b26a621ea42d

8 Disclaimer

Shellboxes reports should not be construed as ”endorsements” or ”disapprovals” of particular teams or projects. These reports do not reflect the economics or value of any ”product” or ”asset” produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology's proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don't offer any kind of investing advice and shouldn't be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

SHELLBOXES

For a Contract Audit, contact us at contact@shellboxes.com