VIDMA

XEND Finance

SMART CONTRACT AUDIT

Project: Xend Finance
Date: May 18th, 2022

TABLE OF CONTENTS

SUMMARY

Vidma team has conducted a smart contract audit for the given codebase.

During the auditing process, the Vidma team has found 37 issues, including issues with critical severity. Worse mention that was found 10 critical issues, but all of them were successfully resolved by the Xend Finance team.

A detailed summary of the issues and their current state is displayed in the table below.

Severity of the issue	Total found	Resolved	Not valid	Unresolved
Critical	11 issues	10 issues	1 issue	0 issues
High	6 issues	6 issues	0 issues	0 issues
Medium	3 issues	3 issues	0 issues	0 issues
Low	7 issues	7 issues	0 issues	0 issues
Informational	10 issues	9 issues	1 issue	0 issues
Total	**37 issues**	**35 issues**	**2 issues**	**0 issues**

After evaluating the findings in this report and the final state after fixes, the Vidma auditors can state that the contract is fully operational and secure. Under the given circumstances, we set the following risk level:

Our auditors are evaluating the initial commit given for the scope of the audit and the last commit with the fixes. This approach helps us adequately and sequentially evaluate the quality of the code. Code style, optimization of the contracts, the number of issues, and risk level of the issues are all taken into consideration. The Vidma team has developed a transparent scoring system presented below.

Severity of the issue	Resolved	Unresolved
Critical	1	10
High	0.8	7
Medium	0.5	5
Low	0.2	0.5
Informational	0	0.1

Please note that the points are deducted out of 100 for each and every issue on the list of findings (according to the current status of the issue). Issues marked as "not valid" are not subject to point deduction.

Based on the **overall result of the audit**, the Vidma audit team grants the following score:

In addition to manual check and static analysis, the auditing team has conducted a number of integrated autotests to ensure the given codebase has an adequate performance and security level.

The test results and the coverage can be found in the accompanying section of this audit report.

Please be aware that this audit does not certify the definitive reliability and security level of the contract. This document describes all vulnerabilities, typos, performance issues, and security issues found by the Vidma audit team. If the code is still under development, we highly recommend running one more audit once the code is finalized.

SCOPE OF WORK

XEND Finance

Credit Unions, Cooperatives, and Individuals anywhere in the world can now earn higher interests in stable currencies on their savings.

Within the scope of this audit, two independent auditors thoroughly investigated the given codebase and analyzed the overall security and performance of the smart contracts.

The audit was conducted from March 1st to May 18th, 2022.The outcome is disclosed in this document.

The scope of work for the given audit consists of the following contracts:

- StakingV2.

The source code was taken from the following **source**:
https://github.com/Cyber-Spawns/staking

Initial commit submitted for the audit:
3988c43700c076bba15cc3e4d64ed0127a0bdf6e

Last commit of the contract:
4f696abc6c1e93bdeecda6d6f392a387d987dbdd

Last commit of the off-chain script:
4d7fb4eaa5fa9f99e3a12a71b85cccd94ccc7b41

As a reference to the contracts logic, business concept, and the expected behavior of the codebase, the Xend Finance team has provided the following documentation:
https://drive.google.com/drive/folders/1ZxqTao1fuq41rnr9BbMDV4wdkxWEWL9V?usp=sharing

WORKFLOW OF THE AUDITING PROCESS

Vidma audit team uses the most sophisticated and contemporary methods and well-developed techniques to ensure contract is free of vulnerabilities and security risks. The overall workflow consists of the following phases:

Phase 1: The research phase

Research

After the Audit kick-off, our security team conducts research on the contract's logic and expected behavior of the audited contract.

Documentation reading

Vidma auditors do a deep dive into your tech documentation with the aim of discovering all the behavior patterns of your codebase and analyzing the potential audit and testing scenarios.

The outcome

At this point, the Vidma auditors are ready to kick off the process. We set the auditing strategies and methods and are prepared to conduct the first audit part.

Phase 2: Manual part of the audit

Manual check

During the manual phase of the audit, the Vidma team manually looks through the code in order to find any security issues, typos, or discrepancies with the logic of the contract. The initial commit as stated in the agreement is taken into consideration.

Static analysis check

Static analysis tools are used to find any other vulnerabilities in smart contracts that were missed after a manual check.

The outcome

An interim report with the list of issues.

Phase 3: Testing part of the audit

Integration tests

Within the testing part, Vidma auditors run integration tests using the Truffle or Hardhat testing framework. The test coverage and the test results are inserted in the accompanying section of this audit report.

The outcome

Second interim report with the list of new issues found during the testing part of the audit process.

STRUCTURE AND ORGANIZATION OF THE FINDINGS

For simplicity in reviewing the findings in this report, Vidma auditors classify the findings in accordance with the severity level of the issues. (from most critical to least critical).

All issues are marked as “Resolved” or “Unresolved”, depending on if they have been fixed by Xend Finance or not. The issues with “Not Valid” status are left on the list of findings but are not eligible for the score points deduction.

The latest commit with the fixes reviewed by the auditors is indicated in the “Scope of Work” section of the report.

The Vidma team always provides a detailed description of the issues and recommendations on how to fix them.

Classification of found issues is graded according to 6 levels of severity described below:

Critical

The issue affects the contract in such a way that funds may be lost or allocated incorrectly, or the issue could result in a significant loss.

Example: Underflow/overflow, precisions, locked funds.

High

The issue significantly affects the ability of the contract to compile or operate. These are potential security or operational issues.

Example: Compilation errors, pausing/unpausing of some functionality, a random value, recursion, the logic that can use all gas from block (too many iterations in the loop), no limitations for locking period, cooldown, arithmetic errors which can cause underflow, etc.

Medium

The issue slightly impacts the contract's ability to operate by slightly hindering its intended behavior.

Example: Absence of emergency withdrawal of funds, using assert for parameter sanitization.

Low

The issue doesn't contain operational or security risks, but are more related to optimization of the codebase.

Example: Unused variables, inappropriate function visibility (public instead of external), useless importing of SCs, misuse or disuse of constant and immutable, absent indexing of parameters in events, absent events to track important state changes, absence of getters for important variables, usage of string as a key instead of a hash, etc.

Informational

Are classified as every point that increases onboarding time and code reading, as well as the issues which have no impact on the contract's ability to operate.

Example: Code style, NatSpec, typos, license, refactoring, naming convention (or unclear naming), layout order, functions order, lack of any type of documentation.

MANUAL REPORT

The user lost his rewards due to wrong saving of unclaimed data.

Critical | Resolved

If the user calls an unstake function when treasure is less than the user's rewards then the remainder of user's rewards saves to unclaimed mapping. But in this case function unstake always sets zero value. It is because at first it equates rewards to treasury value and after this calculates the difference between them so it always returns 0.

```
reward = treasury;
unclaimed[msg.sender] = reward.sub(treasury);
```

Recommendation:

Swap lines

```
reward = treasury;
```

and

```
unclaimed[msg.sender] = reward.sub(treasury);
```

User lost his rewards when called unstake function if treasury value is less than user's rewards

Critical | Resolved

If the value of treasury is less than the amount of user's rewards it sends as many rewards as it can and deletes user from staking. After that user can't get his rewards back.

```
if (reward > treasury) {
      reward = treasury;
      treasury = 0;
    } else {
      treasury = treasury.sub(reward);
    }
```

Recommendation:

Add a mechanism for saving the remaining rewards of the user. In order for the user to be able to get his rewards after the owner adds tokens.

Re-Audit:

After the review of the fixes, this issue is partially fixed as it led to a new issue described in the list below (Critical: The user can unsatake all tokens of the staking contract).

The user can unsatake all tokens of the staking contract

Critical | Resolved

Function “unstake” stopped deleting user information to save information about rewards after the last changes. Because it amount of staked tokens by the user doesn’t change that means that he can unstake as many times as he wants.

Also, it provokes the next issues:

1) The contract continues to add rewards to the user;
2) Wrong total Weight Score calculation;
3) Wrong returned data by function *getTotalStaked(address)*.

Recommendation:

Return deleting of user’s information.

Unexpected unstake opportunity for the stakes with lockups

Critical | Resolved

In the *unstake()* function, there is no check about the lock period of the user’s staked amount, so there is a possibility to unstake lock-up staking without calling *forceUnlock()* but calling simple *unstake()* function. In this case, user will receive his staked funds back without paying the reduction.

Recommendation:

Consider validating if the staking has lockups when *unstake()* function is called.

Unstable logic of pool updating

Critical | Resolved

If off chain script will not be able to run every week it may cause wrong updating of *totalWeightedScore array*. It is because the function *updatePool* takes only one value and sets it for all weeks from last to current only once.

Because off chain script calls this function several times but only the first time will work. It is because of cycle's restriction:

```
uint256 last =
lastRewardBlockTime.sub(startBlockTime).div(ONE_WEEK);
    uint256 current =
block.timestamp.sub(startBlockTime).div(ONE_WEEK);
    for (uint256 i = last; i < current; i++) { …
```

Cycle doesn't allow updating the *totalWeightedScore* array if function calls several times in one week.

For example:
If we didn't run the off chain script for 5 weeks and run it, the off chain script called function *updatePool* 5 times with different values but only the first call worked correctly.

If we haven't called off the chain script from the start of staking.

Week	How should works	How works
0	0	0
1	115384608000	0
2	230769216000	0
3	346153824000	0
4	461538456000	0

If we haven't called off the chain script from some date.

Week	How should works	How works
10	1153846152000	1153846152000
11	1153846152000	1153846152000
12	1269230760000	1153846152000
13	1384615368000	1153846152000
14	1615384608000	1153846152000

Recommendation:

Change logic of off chain script and change *updatePool* function to allow two parameters: array of *totalWeightedScores* and array of weeks.

Example:

```
function updatePool(
        uint256[] _totalWeightedScoreArray,
        uint256[] _weekArray
    ) external onlyGovernment {
        require(block.timestamp >= lastRewardBlockTime, "invalid
call");
        require(
            _totalWeightedScoreArray.length == _weekArray.length,
            "length mismatch"
        );
        uint256 last =
lastRewardBlockTime.sub(startBlockTime).div(ONE_WEEK);
        uint256 current =
block.timestamp.sub(startBlockTime).div(ONE_WEEK);
        for (uint256 i = 0; i < _weekArray.length; i++) {
            if (last == 0) break;
            rewardDrop[_weekArray[i]] = rewardDrop[_weekArray[i] -
1].sub(
                rewardDrop[_weekArray[i] - 1].div(100)
            );
            totalWeightedScore[_weekArray[i]] =
```

```
_totalWeightedScoreArray[i];
        }
        lastRewardBlockTime = block.timestamp;
    }
```

StakingV2 contract doesn't have logic of reserving tokens

Critical | Resolved

User's stake tokens and get additional tokens as rewards while staking but the contract doesn't track the amount of tokens on his balance. Because of this, users can unstake tokens with rewards and the contract doesn't have enough tokens for unstaking by another user.

For example, two users stake 1000 tokens. After some time they have some amount of rewards e.g. 0,0001 token. One user unstake his tokens with rewards and contract have less than 1000 tokens (999,999) after that user #2 can't withdraw his tokens because of error *ERC20: transfer amount exceeds balance*.

Recommendation:

Add function *addTokens* or *addLiquidity* for Owner that allows to transfer tokens from Owner to the contract.

Add require that user can't stake if contract's balance is less than *totalStaked*2*.

The owner can set zero *rewardDrop* value

Critical | Resolved

The owner can pass any value of *rewardsDrop* in the constructor. If he passes zero this means that rewards will always be equal to 0.

Recommendation:

Add requirements for preventing passing zero value of *rewardsDrop*.

User who starts staking later than others get the same amount of rewards

Critical | Resolved

For example: if the first user stakes 1000 tokens and after 6 months the second user stakes the same amount of tokens they get the same amount of reward tokens.

This shouldn't work like this, the first user must have more reward tokens than the second user.

It works wrong because calculation of rewards tokens doesn't use variables *stakeTime[]*.

Recommendation:

Change calculation of rewards.

Wrong declared variable in inner *for* cycle

Critical | Resolved

Commit e8e4314efb70fa1d3b365eb9d0687bdf1ccb3eac.

```
for (let i = lastWeekNumber; i < currentWeek; i++) {
    let totalScore = 0
    for (let i = 0; i < lengthOfStakers; i++) {
        const staker = await contract.methods.stakers(i).call()
        const score = await
        contract.methods.getWeightedScore(staker, i)
```

External and inner cycles use the same variable.

Recommendation:

Change variable name in inner cycle e.g. on *j* and use it for *staker(j).call()* method.

Re-Audit:

Is fixed in commit "c8eb0cd5022c76a25b58dfac092fdadacdee5bde".

Possibility to combine standard staking and lockups

Critical | Resolved

According to the documentation, there should be not any possibility for a single wallet to combine standard stakings and lockups. But in the contract it is possible. A User can stake funds with a lockup period for example 3 months and he will be able to add another staking via the next transactions but without lockups only (standard stakings). It is possible only in that case when the first user staking will be with lockups and all next without.

This leads to the other critical issue related to the *forceUnlock()* and *unstake()* execution after the user will have combined staking at the same time. When the user has combined staking and will try to call the *forceUnlock()* function he won't be able to unstaked after his funds because in the *forceUnlock()* function all info about stakingIds is deleted in line 184. Hence user's funds from the standard staking will be locked in the contract's balance without any chance to withdraw it.

Steps to reproduce:

1) Alice call *stake(1000, THREE_MONTHS)*;
2) Alice call *stake(500, NO_LOCK)*;
3) State was changed:
   getStakedAmount(Alice) -> 1500;
   getStakeAmount() -> 1500;
3) Alice call *forceUnlock()*;
3) State was changed:
   getStakedAmount(Alice) -> 0;
   getStakeAmount() -> 500;
3) Alice call *unstake()* and nothing is transfering to the Alice.

Recommendation:

Consider adding more strict validation for staking locking types when the user makes multiple staking.

When *forceUnlock()* is called remaining reward isn't claiming

Critical | Not valid

When the user call *forceUnlock()* remaining reward isn't transferring to the user and he will not be able to claim it after because all info regarding this stakeId is deleted from *userInfo[user]* mapping.

Recommendation:

Consider transferring the remaining reward to the user when *forceUnlock()* is called.

Re-Audit:

Explanation: Xend Finance team intended to remove reward when users try to unlock until lockup ends because reward was generated from lockup benefits.

Risk of overflow max js int value

High | Resolved

Max js value equal to 2^53-1 or 9007199254740991. It means that when an off-chain script converts value from BN to Number it can cause an error "Number can only safely store up to 53 bits". It will happen only if total staked tokens are equal to or bigger than *1_501_199_875_791* tokens.

Recommendation:

Saves scores like BN or string types.

User can create one more active staking than *maxActiveStake*

High | Resolved

Require:

```
require(stakingIds.length <= maxActiveStake, "exceed
maxActiveStake");
```

Allows to create another one stake because of operator <=.

Recommendation:

Change <= to <.

Overflow risk

High | Resolved

In the Solidity versions, less than 0.8.x, integers proceed overflow and underflow without any errors. In line 120 of the Stake contract, the counter is incremented with a simple “+” operator which can lead to the overflow when the value reaches the max available number.

Recommendation:

Use *add()* function from SafeMath library to deal with arithmetic addition operation.

Function *apy()* always returns 0

High | Resolved

Then *totalStaked != 0* the function always returns 0.

It is because variable *rewardDrop[current]* is always equal to 0.

It is because, *updatePool* function never sets the value of *rewardDrop[current]*.

```
for (uint256 i = last; i < current; i++) {
    if (last == 0) break;
    rewardDrop[i] = rewardDrop[i-1].sub(rewardDrop[i-1].div(100));
    totalWeightedScore[i] = _totalWeightedScore;
    emit ShowTheFuckingRewardDrop(rewardDrop[i],
    rewardDrop[i-1],totalWeightedScore[i],i);
}
```

Recommendation:

Change calculations of *apy* from

```
_apy = rewardDrop[current].mul(WEEKS_OF_ONE_YEAR).mul(MAX_BPS).
div(totalStaked);
```

to

```
_apy = rewardDrop[current-1].mul(WEEKS_OF_ONE_YEAR).mul(MAX_BPS).
div(totalStaked);
```

Using the wrong operator for *BigNumber* object

High | Resolved

Commit e8e4314efb70fa1d3b365eb9d0687bdf1ccb3eac.

```
let totalScore = 0
    for (let i = 0; i < lengthOfStakers; i++) {
        const staker = await contract.methods.stakers(i).call()
        const score = await
        contract.methods.getWeightedScore(staker, i)
        totalScore += score
        await new Promise(resolve => setTimeout(resolve, 1 * 1000));
    }
```

getWeightedScore function returns BN object so you need to use bn.js for correct calculations.

Using simple operations can provoke next error:

Error: invalid BigNumber string (argument="value", value="0NaN", code=INVALID_ARGUMENT, version=bignumber/5.4.1)

Recommendation:

import bn.js in your script, declare *totalScore* as BN object and add score to *totalScore* by adding function of bn.js.

Example:

```
let totalScore = new BN("0");
for (let j = 0; j < lengthOfStakers; j++) {
    const staker = await contract.methods.stakers(j).call()
    const score = await contract.methods.getWeightedScore(staker, i)
    totalScore = totalScore.add(new BN(score.toString()));
    await new Promise(resolve => setTimeout(resolve, 1 * 1000));
    // delay 10s
}
```

Incorrect tracking of valid stakers

High | Resolved

When the user stakes he is added to the stakers array. But it is happening every time when the *stake()* function is called, so if the user makes multiple staking he is adding every time to the stakers array. It can lead to the critical issue related to the script that makes some computation to calculate *_totalWeightedScore*. When the same staker address will be not unique in the stakers array the *_totalWeightedScore* will be calculated incorrect (*_totalWeightedScore* is accumulating amount of weighted score of each staker based on the week number).

Another point is when the user *unstake()* he is not deleting from the stakers array which can mislead the users about the current state of valid stakers.

Recommendation:

Consider updating stakers array appropriate.

Wrong output in *pendingReward* function

Medium | Resolved

Pending rewards of the caller subtract from rewards of the passed user:

```
function pendingReward(address _user) external view returns
(uint256) {
    return
_pendingReward(_user).sub(userInfo[msg.sender].rewardDebt);
  }
```

Recommendation:

Change *msg.sender* to *_user*.

Inappropriate state variable initialization in the loop

Medium | Resolved

In the function *updatePool()* there is a loop for iteration among past weeks and the value of *totalWeightedScore* for the current week is initialized inside the loop (line 196). It is inappropriate as it can be done out of the loop and reduce gas cost for the transaction execution.

Recommendation:

Consider moving initialization of *totalWeightedScore[current]* out of loop.

Lack of validation of reduction percent in the appropriate setter function

Medium | Resolved

In the function *setReductionPercent()* there is no check if the new percent is less than 100% which can lead to incorrect arithmetic calculation in the *forceUnlock()* function.

Recommendation:

Consider adding validation for new percent value.

Function *setRewardDrop* misses event

Low | Resolved

Function *setRewardDrop* doesn't have an event for notification about rewardDrop updating.

Recommendation:

Add an event which will describe changing of rewardDrop.

Inappropriate initialization of *lastUnstakeTime*

Low | Resolved

The *stake()* function is tracking the time of the last unstake time in line 152 which is useless because this information is deleted in the next line (Line 153). Hence *require* in Line 139 is also useless because *userInfo[msg.sender].lastUnstakeTime* will always equal “0” and as docs say partly unstake is not allowed.

Recommendation:

Consider removing useless initialization and require to decrease gas usage for the transaction call.

Lack of events emission while the critical data state is changing

Low | Resolved

In function *transferGovernance()* critical access control parameter of government is changed and in the function *setReductionPercent()* important arithmetic parameter is changing. There is no event emission for these changes to track off-chain change.

Recommendation:

Emit an event for critical parameter changes in both functions *transferGovernance()* and *setReductionPercent()*.

Division by zero

Low | Resolved

- When *totalStaked* variable equals zero *apy()* function return error “Division by zero”;
- When *weightedScore* variable equals zero *_pendingReward()* function return error “Division by zero”.

Recommendation:

- Consider adding *require* where check the *totalStake* amount in the *apy()* function;
- Consider managing case when the *weightedScore* equals zero while the reward is calculating in the *_pendingReward()* function.

Lack of zero-check

Low | Resolved

In functions *transferGovernance()* there is no check if the *_newGov* address isn't a zero address.

Recommendation:

Consider adding a check for zero address for function *transferGovernance()*.

Unspecified state variable visibility

Low | Resolved

It is best practice to set the visibility of state variables explicitly. The default visibility for *scoreLevels* in line 58 is internal.

Recommendation:

Explicitly specify the visibility of state variables.

Not indexed parameters in events

Low | Resolved

In events of the Stake contract, there are any indexed parameters. The indexed parameters for logged events give the possibility to search for these events using the indexed parameters as filters.

Recommendation:

Consider adding *indexed* keyword to the parameters in the events (possibly up to three indexed parameters).

Incorrect annotation of *apr()* function

Informational | Resolved

Annotation of function *apr* describing APY but must be about APR.

Recommendation:

Change APY and yield to APR and rate.

Unused state variable

Informational | Resolved

- State variable *ONE_YEAR* is never used in the Stake contract;
- Struct PoolInfo has an unused field *accRewardPerShare*.

Recommendation:

Consider removing unused state variables.

Wrong passed data to *ForceUnlock* event

Informational | Resolved

```
token.safeTransfer(msg.sender, deposits[stakingId].sub(reduction));

// update the state variables
totalStaked = totalStaked.sub(deposits[stakingId]);
deposits[stakingId] = 0;
delete userInfo[msg.sender];

emit ForceUnlock(msg.sender, stakingId, deposits[stakingId],
lockPeriod[stakingId], offset);
```

This event always returns the parameter *amount* with zero value. Because it emits after changing *deposits[stakingId]*.

Recommendation:

Pass value *deposits[stakingId]* before changing.

EmergencyWithdraw event isn't used

Informational | Resolved

The *EmergencyWithdraw* event isn't used by any function of the StakingV2 contract.

Recommendation:

Delete *EmergencyWithdraw* event.

The user isn’t deleted after *forceUnlock*

Informational | Resolved

After the *forceUnlock* function contract deletes information about user but doesn’t delete him from the list of stakers.

Recommendation:

Add deleting user from the list of stakers.

Lack of NatSpec annotations

Informational | Resolved

Smart contract Stakel is not covered by NatSpec annotations.

Recommendation:

Consider covering by NatSpec all contract methods.

The contract’s name does not match with the filename

Informational | Resolved

Filename StakingV2.sol doesn't match with the contract name Stake. According to the style guide contract, contract and library names should also match their filenames.

Recommendation:

Consider keeping the same name for the contract and the filename.

Some kind of typos in the code source

Informational | Resolved

There are some typos in modifier and variable names:

- Line 42 *_governmant* instead of *_government*;
- Line 67 *onlyGovernmant* instead of *onlyGovernment*.

Recommendation:

Consider fixing typos.

Lack of check whether the staker is valid in *unstake()* function or not

Informational | Resolved

In the *unstake()* function there is no check if the user has any valid stakes. That's why the user who is not a staker can call the unstake function.

Recommendation:

Consider adding a check if the user has any available amount to unstake.

Useless initialization

Informational | Resolved

In the constructor, there is the initialization of the *counter* variable as "0". It is useless as integers are initialized as "0" by default.

Recommendation:

Consider removing initialization of *counter* from the constructor.

User can’t extend the lockup period

Informational | Not valid

Your docs has the following lines: “Users can extend the lockup period to the higher (for example, change lockup vault to 12 months instead of 3, even if it's the 2nd month), but can't shorten it.”

But the contract doesn’t have this functionality.

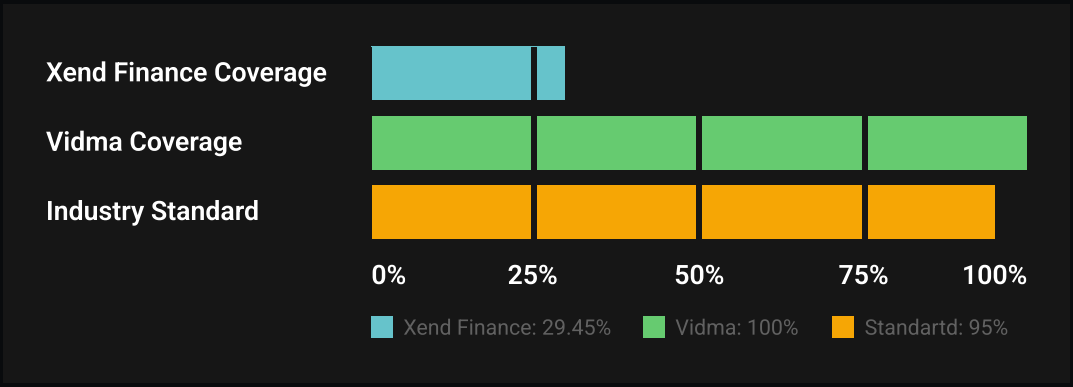
Recommendation:

Add this functionality.

TEST RESULTS

To verify the security of the contract and the performance, a number of integration tests were carried out using the Truffle testing framework.

In this section, we provide both tests written by Xend Finance and tests written by Vidma auditors.



It is important to note that Vidma auditors do not modify, edit or add tests to the existing tests provided in the Xend Finance repository. We write totally separate tests with code coverage of a minimum of 95% to meet the industry standards.

Tests written by Xend Finance

Test Coverage

File	% Stmts	% Branch	% Funcs	% Lines
contracts\	29.45	7.69	13.04	29.05
StakingV2.sol	29.45	7.69	13.04	29.05
All files	**29.45**	**7.69**	**13.04**	**29.05**

Test Results

```
Contract: StakingV2
  ✓ should distribute tokens properly for each staker (4085ms)

1 passing (4085ms)
```

Tests written by Vidma

Test Coverage

File	% Stmts	% Branch	% Funcs	% Lines
contracts\	100	100	100	100
StakingV2.sol	100	100	100	100
All files	**100**	**100**	**100**	**100**

Test Results

```
Contract: StakingV2
  Constructor
    ✓ Must fail if set reward drop to 0 (1138ms)
    ✓ Check correct deploy (1153ms)
  stake
    ✓ Must fail if amount equal to zero (282ms)
    ✓ Must fail if pass wrong lock period (785ms)
    ✓ Must create staking correctly (5043ms)
    ✓ Must fail if try to create staking in short period (297ms)
    ✓ Must fail if try to stake lock vault and standard
      vault (384ms)
    ✓ Must fail if try to stake more times than maxActiveStake
      vault (741ms)
    ✓ Must stake standard wallet correctly second time after
      1 day (1645ms)
    ✓ Must create staking with lock period correctly (1284ms)
    ✓ Must fail if try to create staking with standart
      vault after creating staking with lock vault (370ms)
    ✓ Must fail if try to stake second lock vault (331ms)
    ✓ Must fail if try to stake second lock vault with
      another lock period (291ms)
  pendingReward
```

```
  ✓ Must return zero if first week hasn't passed (266ms)
  ✓ Must return zero if pool hasn't been updated (450ms)
  ✓ Must return some rewards after pool updating (11661ms)
  ✓ Must update rewards even if hasn't been called function
    updatePool (3566ms)
  ✓ Must continue to add rewards after 12th month (124498ms)
  ✓ Must calculate rewards between users correctly (18738ms)
  ✓ Must calculate rewards between users with different
    staking time correctly (23414ms)
  ✓ Must calculate reward calculated relatively to MIN_APR (29889ms)
  ✓ Must calculate reward calculated relatively to MAX_APR (14855ms)
unstake
  ✓ Set up env (3920ms)
  ✓ Must fail if sender hasn't staked yet (307ms)
  ✓ Must fail if min lock period hasn't passed yet (382ms)
  ✓ Must fail if lock period hasn't passed yet (357ms)
  ✓ Must unstake tokens correctly if treasury equal to 0 (11158ms)
  ✓ Must unstake tokens correctly with claiming rewards (2932ms)
  ✓ Must unstake tokens correctly after unstaking by another
    user (1796ms)
  ✓ Must allow to unstake after lock period end (8212ms)
  ✓ Mustn't allows to unstake several times (15303ms)
claimReward
  ✓ Must fail if user doesn't have available rewards (2174ms)
  ✓ Must claim rewards correctly (22714ms)
  ✓ Must claim less rewards than expected because of small
    treasury value (2067ms)
  ✓ Must fail to claim unclaimed rewards if treasure equal to 0
    or less than unclaimed rewards (3202ms)
  ✓ Must claim rewards after unstake with unclaimed rewards
    rewards (1152ms)
forceUnlock
  ✓ Set up (2474ms)
  ✓ Must fail if sender isn't depositor (330ms)
  ✓ Must fail if try to unlock before ending of min lock
    time period (340ms)
  ✓ Must fail if staking isn't locked (797ms)
  ✓ Must force unlock correctly (2695ms)
  ✓ Must stop add rewards after force unlock (4863ms)
setLockTime
  ✓ Must fail if sender isn't owner (202ms)
  ✓ Must fail if pass zero value (189ms)
```

```
  ✓ Must change lockTime correctly (689ms)
setReductionPercent
  ✓ Must fail if sender isn't governance (249ms)
  ✓ Must fail if pass value bigger than MAX_BPS (205ms)
  ✓ Must change lockTime correctly (495ms)
transferGovernance
  ✓ Must fail if sender isn't owner (259ms)
  ✓ Must fail if pass zero address (233ms)
  ✓ Must change lockTime correctly (777ms)
setActionLimit
  ✓ Must fail if sender isn't owner (232ms)
  ✓ Must fail if pass zero value (331ms)
  ✓ Must change action limit correctly (711ms)
setRewardDrop
  ✓ Must fail if sender isn't owner (707ms)
  ✓ Must fail if contract doesn't have staked tokens (255ms)
  ✓ Must fail if owner try to set percent bigger than MAX_APR
    or less than MIN_APR (1594ms)
  ✓ Must update reward drop correctly (824ms)
apr
  ✓ Must return MAX_APY if totalStaked equal to 0 (255ms)
  ✓ Must return MIN_APR correctly (5445ms)
  ✓ Must return MAX_APR correctly (3867ms)
updatePool
  ✓ Must return MAX_APR correctly (3867ms)
  ✓ Must fail if sender isn't government (246ms)
  ✓ Must fail if the function has called before lastRewardBlock
    timestamp (311ms)
  ✓ Must update rewardDrop correctly with zero staked
    tokens (2104ms)
  ✓ Must update rewardDrop correctly with staked tokens (4215ms)
  ✓ Must update scores several times without staked
    tokens (4131ms)
  ✓ Must update scores several times with time interval without
    staked tokens (6599ms)
  ✓ Must update totalScore correctly after 1 year (7512ms)
setMaxActiveStake
  ✓ Must fail if caller isn't owner (225ms)
  ✓ Must fail if pass zero value (193ms)
  ✓ Must change max active stake correctly (470ms)
addReward
  ✓ Must fail if caller isn't owner (211ms)
```

```
    ✓ Must fail if transferred amount exceeds balance (321ms)
    ✓ Must change max active stake correctly (653ms)

77 passing (8m)
```

We are delighted to have a chance to work with the Xend Finance team and contribute to your company's success by reviewing and certifying the security of your smart contracts

The statements made in this document should be interpreted neither as investment or legal advice, nor should its authors be held accountable for decisions made based on this document.

Vidma is a security audit company helping crypto companies ensure their code and products operate safely and as intended, enabling founders to sleep soundly at night. We specialize in auditing DeFi protocols, layer one protocols, and marketplace solutions. Our team consists of experienced and internationally trained specialists. Our company is based in Ukraine, known for its strong engineering, cryptography, and cybersecurity culture.

Website: **vidma.io**
Email: **security@vidma.io**