Table of Contents

Introduction

A time-boxed security review of the **Spillways**' staking contract, with a focus on smart contract security, architecture, and gas optimizations.

Author: **Jacopod**, an independent security researcher.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time and resource-bound effort to find as many vulnerabilities as possible, but there is no guarantee that all issues will be found. A security researcher holds no responsibility for the findings provided in this document. A security review is not an endorsement of the underlying business or product and can never be used as a guarantee that the protocol is bug-free. This security review is focused solely on the security aspects of the Solidity implementation of the contracts. Gas optimizations are not the main focus, but significant inefficiencies will also be reported.

Protocol Summary

Spillways staking is a staking contract where the Spillway token can staked in exchange for another ERC20 token as a reward (USDC). Staking rewards are distributed in epochs, that are intended to last for 7 days. The amount of rewards distributed as well as the exact time of distribution are chosen by the contract owner at the end of each epoch.

An antibot logic is implemented to prevent bots from staking just before the rewards distribution and unstaking right after. The logic consists of excluding from the rewards distribution those tokens that were staked in the last 2 days of the epoch (configurable).

For more info about Spillways, read their website.

Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	High	High	Medium

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions and the cost of the attack is relatively low to the amount of funds that can be stolen or lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - has too many or too unlikely assumptions or requires a huge stake by the attacker with little or no incentive.

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to unexpected behavior with some of the protocol's functionalities that are not so critical.

Actions required by severity level

- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

Scope

- Date: `2023-12-22`
- Duration of the audit: 7 days
- Audited commit hash: 9f6ce274c7a52268c60aa1ab3b9c180e5bb59b43
- Review Commit hash: [....]
- Number of Solidity Lines Of Code (nSLOC): 219

Inside scope

The following smart contracts were in the scope of the audit:

- `SpillwaysStake`

Outside scope

The staking contract inherits or interacts with the following libraries and smart contracts, but they are not part of the scope:

- `Spillway token` 0x8790f2fc7ca2e7db841307fb3f4e72a03baf7b47 (ethereum mainnet)

- `USDC token` 0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48 (ethereum mainnet)
- `Ownable` (Open Zeppelin V4)
- `ERC20` (Open Zeppelin V4)

Executive Summary

Observations

- **Architecture**: the contract architecture does not have a very gas-efficient design. Despite this not being a vulnerability per se, it can impact the user experience. More specifically, the gas cost of claiming rewards scales linearly with the number of epochs since the last claim. Fixing this would require an in-depth re-resign of the architecture and probably re-writing the contract from scratch.

- **Rewards**: Some issues related to rewards calculations were found.

- **Governance**: Several Governance-related issues make the contract highly centralized, giving significant power to the contract owner. It is worth mentioning that the contract ownership is planned to be transferred to a DAO.

  Governance issues do not immediately mean a risk for the protocol or its users but can be very dangerous if the governance mechanisms are compromised or if the contract owner acts maliciously.

Main Findings Summary

Issue ID	Description	Severity	Fixed
[H-1]	Wrong anti-bot logic implementation causes some rewards to not be distributed, reducing rewards for stakers	High	Acknowledged
[M-1]	Stakers that stake and unstake within a 2-day anti-bot period will lose an unfair amount of rewards	Medium	Acknowledged
[M-2]	The contract `owner` can withdraw all staked tokens and unclaimed rewards	Medium	Acknowledged
[M-3]	Contract owner can stop users from unstaking funds and claiming rewards by pausing the contract	Medium	Acknowledged
[M-4]	Distribution of rewards might be delayed or discontinued if contract ownership is transferred to a DAO	Medium	✓
[M-5]	All public functions will revert due to a lack of reward tokens if the owner withdraws too many reward tokens with `emergencyWithdraw()`	Medium	Acknowledged

Informational and gas optimizations are excluded from this table. All low-severity findings have been acknowledged.

Detailed findings

High

[H-1] Wrong anti-bot logic implementation causes some rewards to not be distributed, reducing rewards for stakers

Description

According to the antibot logic, stakers of the last two days are not entitled to rewards. However, the non-given rewards should be distributed to other stakers or accounted for somehow. This is not the case, as the calculation of `epochRewardRate` uses `totalStakedTokens` as a divisor to calculate how much rewards per spillway tokens are to be given, and `totalStakedTokens` also includes the tokens staked in the last 2 days of the epoch.

```
    function startNewEpoch(uint256 _rewards) external onlyOwner {
        require(_rewards > 0, "Rewards must be greater than 0");
        epochData[currentEpoch].epochEnd = block.timestamp;
        if (totalStaked > 0) {
@>          epochData[currentEpoch].epochRewardRate = (_rewards * 1e18) / 
totalStaked;
        } else {
          ...
```

Therefore, the reward rate is also shared with the staked amounts of the last 2 days of the epoch, but these stakers are not allowed to claim them. Thus, the corresponding rewards will get locked in the contract balance, unallocated to any staker.

Severity classification

- Likelihood: high, as it happens in every epoch where some tokens are staked during the last two days.
- Impact: medium, as the rewards reduction for each staker is not too big, because it is shared among all stakers.
- Overall severity: **high**.

Proof of Concept

A detailed proof of concept written in Foundry can be found below. All the proof of concepts inherit from `BaseForProofOfConcepts` contract, which is a basic setup that can be found in the Appendix.

Click on "details" to see a Proof of Code written in Foundry:

▶ Details

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "lib/forge-std/src/Test.sol";
import {SpillwaysStake} from "src/staking.sol";
import "test/pocs/baseForPoCs.t.sol";

contract TestLostRewardsOfLastTwoDays is BaseForProofOfConcepts {
    address staker1 = makeAddr("staker1");
    address staker2 = makeAddr("staker2");
    address staker3 = makeAddr("staker3");

    function setUp() public override {
        super.setUp();
    }

    function test_stakingRewardsFromLastTwoDaysAreLost() public {
        // lets give some tokens to some potential stakers, and approve the
contract
        deal(address(spilltoken), staker1, 100_000 ether);
        vm.prank(staker1);
        spilltoken.approve(address(staking), type(uint256).max);

        deal(address(spilltoken), staker2, 100_000 ether);
        vm.prank(staker2);
        spilltoken.approve(address(staking), type(uint256).max);

        deal(address(spilltoken), staker3, 100_000 ether);
        vm.prank(staker3);
        spilltoken.approve(address(staking), type(uint256).max);

        // none of the three stakers has any reward token in their wallet
to begin with
        assertEq(usdc.balanceOf(staker1), 0);
        assertEq(usdc.balanceOf(staker2), 0);
        assertEq(usdc.balanceOf(staker3), 0);

        // a new epoch starts. These rewards are irrelevant for the PoC
        vm.prank(owner);
        staking.startNewEpoch(1_000 ether);

        // staker1 and staker2 will unstake together the same as staker3
        vm.prank(staker1);
        staking.stake(50_000 ether);
        vm.prank(staker2);
        staking.stake(50_000 ether);
        // staker3 stakes at the end of the week, only 1 day before the
epoch finishes
        skip(6 days);
        vm.prank(staker3);
        staking.stake(100_000 ether);

        // note that the total staked amount is equal to the three stakers,
so there is nobody else staking
        assertEq(staking.totalStaked(), 200_000 ether);

        // one day later, the owner distributes rewards, and a new epoch
starts
        skip(1 days);
        vm.prank(owner);
        uint256 rewards = 1_000 ether;
        staking.startNewEpoch(rewards);

        // when all stakers claim the rewards, staker3 gets nothing,
because he staked in the antibot period
        vm.prank(staker1);
        staking.claim();
        vm.prank(staker2);
        staking.claim();
        vm.prank(staker3);
        staking.claim();

        uint256 rewardsStaker1 = usdc.balanceOf(staker1);
        uint256 rewardsStaker2 = usdc.balanceOf(staker2);
        uint256 rewardsStaker3 = usdc.balanceOf(staker3);

        uint256 totalRewardsDistributed = rewardsStaker1 + rewardsStaker2 +
rewardsStaker3;

        // THIS ASSERTION FAILS
        assertEq(rewards, totalRewardsDistributed, "Rewards are lost");

        // because staker3 go 0 rewards, and staker 1 and 2 got their share
proportional to totalStaked amount,
        // which includes the stake from staker3.
        assertEq(rewardsStaker1, 250 ether);
        assertEq(rewardsStaker2, 250 ether);
        assertEq(rewardsStaker3, 0);

        // Moreover, after all stakers have claimed, the usdc balance of
the contract is still positive
        // These rewards are not claimable
        assertEq(usdc.balanceOf(address(staking)), 500 ether);
    }
}
```

Recommendation

Instead of discarding the rewards from staked tokens of the last two days of each epoch, buffer those discarded rewards and distribute them in the next epoch.

More specifically, when rewards are calculated in `_calculateRewards`, a certain amount of stake is subtracted from the `stakedAmount`. On the subtracted amount, rewards are calculated and given to the user. Instead of discarding that subtracted amount, use it to calculate how much rewards have been excluded, and buffer them in a state variable. The next time `startNewEpoch()` is called, the buffered rewards can be added to the `rewards` that the owner is distributing before it is divided by `totalStaked`.

Fixes review

The reward lost per staker is not large enough to justify a new implementation, so the team decided to not implement a fix. Moreover, as the team decided to keep the function `emergencyWithdraw()` in the contract, the non-distributed rewards can be rescued, and would therefore not be locked in the contract.

Medium

[M-1] Stakers that stake and unstake within a 2-day anti-bot period will lose an unfair amount of rewards

Description

For users staking and unstaking during the antibot time, the rewards they earn from that epoch are reduced beyond what is fair. This can go as high, as making the staker lose all staking rewards of the epoch.

The `staker.stakeHistory` keeps track of the stakes a user has made and the `staker.totalStakedAmount` keeps track of the total stake.

Every time the `stake()` function is called, a new stake is pushed to `staker.stakeHistory`, and the `staker.totalStakedAmount` is increased accordingly.

However, when the `unstake()` function is called, only the `staker.totalStakedAmount` is reduced, but the stake history remains.

When calculating rewards, the `staker.stakeHistory` is used to check whether a stake happened during the antibot time or not, to remove it from the amount eligible for rewards. As the history is not properly deleted when calling `unstake()`, the tokens that were staked during the antibot time will be subtracted in `_calculateRewards` even if they were previously unstaked.

```
        if (staker.stakeHistory[endEpoch].length > 0 && currentEpoch != 2) {
            for (uint256 i = staker.stakeHistory[endEpoch].length - 1; i >=
0; i--) {
@>              if (staker.stakeHistory[endEpoch][i].timeofStaking <
thresholdTimestamp) {
                    break;
                }
                if (staker.stakeHistory[endEpoch][i].stakedAmount >
stakedAmount) {
                    stakedAmount = 0;
                    break;
                }
@>              stakedAmount = stakedAmount - staker.stakeHistory[endEpoch]
[i].stakedAmount;

                if (i == 0) {
                    break;
                }
            }
        }
```

Severity classification

- Likelihood: medium, as it only affects stakers of the last two days of an epoch, for the rewards of that epoch.
- Impact: high, as they can lose up to all rewards of the epoch
- Overall severity: **medium**

Proof of Concept

A user stakes a certain unit (an arbitrary amount of spillways) before the antibot time starts, and the same amount once the antibot time starts. An array of two stakes is stored in `staker.stakeHistory`, and the `staker.totalStakedAmount = 2 * unit`.

If the user unstakes 1 unit, the `staker.totalStakedAmount = 1 unit`, which was staked before the antibot time started.

However, when calling `claim()`, the antibot logic will iterate the array of stakes from the back, subtracting each `stakedAmount` from the antibot period, which will not be eligible for rewards. As the user staked 1 unit during the antibot period, but this stake is not deleted when unstaking, the `stakedAmount` resulting inside `_calculateRewards` will be exactly 0, therefore not earning rewards for the initial unit they staked before the antibot time. Click on "details" to see a Proof of Code written in Foundry:

▶ Details

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "lib/forge-std/src/Test.sol";
import {SpillwaysStake} from "src/staking.sol";
import "test/pocs/baseForPoCs.t.sol";

contract TestStakingSequenceForLostRewards is BaseForProofOfConcepts {
    address staker1 = makeAddr("staker1");
    address staker2 = makeAddr("staker2");
    address staker3 = makeAddr("staker3");

    function setUp() public override {
        super.setUp();
    }

    function test_specificStakingSequenceThatCausesRewardsLost() public {
        // lets give some tokens to some potential stakers, and approve the
contract
        deal(address(spilltoken), staker1, 1_000_000 ether);
        vm.prank(staker1);
        spilltoken.approve(address(staking), type(uint256).max);
        // staker has 0 rewards initially
        assertEq(usdc.balanceOf(staker1), 0);

        // a new epoch starts. These rewards are irrelevant for the PoC
        vm.prank(owner);
        staking.startNewEpoch(1_000 ether);
        uint256 stakingUnit = 10_000 ether;

        // staker stakes one staking unit before the antibott time starts
        skip(1 days);
        vm.prank(staker1);
        staking.stake(stakingUnit);

        // when the antibot time starts, the same staker stakes another
unit and stakes it again within the antibot time
        skip(5 days);
        vm.prank(staker1);
        staking.stake(stakingUnit);
        vm.prank(staker1);
        staking.unstake(stakingUnit);

        // The owner starts a new epoch just one day after so that the two
last operations fall inside the antibot time
        vm.prank(owner);
        staking.startNewEpoch(1_000 ether);

        //The staker claims rewards from the last epoch but receives
NOTHING even though the first stake was done
        // before the antibot time started
        vm.prank(staker1);
        staking.claim();
        assertEq(usdc.balanceOf(staker1), 0);
    }
}
```

Recommendation

Unfortunately, a fix to this problem is not trivial with the current implementation, as unstaking partial amounts from previous stakes is not contemplated.

A possible idea would be to temporarily disable `unstake()` for users that have staked during the anti-bot period until the epoch finishes, but that is by far not an ideal solution, as a user should always be able to withdraw their funds.

Fixes review

The team acknowledged the issue, but the impact was not considered high enough to justify a fix. The non-distributed rewards due to bad accounting will sit in the contract until `emergencyWithdraw()` is called by the owner.

[M-2] The contract `owner` can withdraw all staked tokens and unclaimed rewards

Description

Stakers can lose all their staked funds and unclaimed rewards if the contract owner acts maliciously.

The function `emergencyWithdraw()` allows the contract `owner` to transfer out of the contract any amount of any ERC20 token. This includes the staked Spillway tokens and the USDC rewards that haven't been claimed yet.

Severity classification

- Likelihood: low, as the contract owner (or DAO) is expected to act in good faith.
- Impact: high, as stakers could lose all their funds and rewards.
- Overall severity: **medium**.

Recommendation

Stakers should be the only ones authorized to move their tokens. The first recommendation is to remove the `emergencyWithdraw()` function from the contract. Even though the contract owner is willing to act in good faith, the stakers are exposed to an unnecessary risk. Moreover, even if the ownership is transferred to a DAO, the stakers are still exposed in the case that the DAO is compromised or acts maliciously.

If the first recommendation is denied, my second recommendation is to at least restrict the function to not be able to withdraw staked funds, and only reward tokens.

```
    function emergencyWithdraw(address token, uint256 amount, address _to) external onlyOwner {
+       require(token != address(spillToken), "Spillway tokens cannot be withdrawn");
        require(IERC20(token).transfer(_to, amount), "Emergency withdraw failed");
    }
```

Fixes review

The team has decided to keep the `emergencyWithdraw()` function without any changes and transfer the ownership to a DAO.

[M-3] Contract owner can stop users from unstaking funds and claiming rewards by pausing the contract

Description

Stakers will not be able to unstake their tokens nor claim their rewards if the contract owner acts maliciously and pauses the contract indefinitely.

The `Active` modifier reverts when `_paused = true`. This modifier is used in the functions `stake()`, `unstake()`, and `claim()`. The contract owner is the one that can pause the contract by setting `_paused = true`.

This means that if the contract owner pauses the contract, users will not be able to withdraw their spillway tokens, nor claim their rewards.

Severity classification

- Likelihood: low, as the contract owner (or DAO) is expected to act in good faith.
- Impact: high, as stakers could lose access to their funds and rewards.
- Overall severity: **medium**.

Recommendation

Remove the `Active` modifier for `unstake()` and `claim()` functions. Using it in `stake()` is OK, because stopping users from staking more funds is acceptable, but the users should always be able to withdraw their funds or their rewards.

Fixes review

No fixes applied. The team considers it safe enough if the contract ownership is transferred to a DAO.

[M-4] Distribution of rewards might be delayed or discontinued if contract ownership is transferred to a DAO

Description

Delegating the responsibility of calling time-sensitive functions to a DAO can be very impractical, as reaching a consensus can take time. If the contract ownership is transferred to a DAO as the team intends, time-sensitive operations like calling `startNewEpoch()` could be significantly delayed if consensus is not reached by the DAO. This can delay or discontinue the rewards distribution, directly breaking one of the core invariants of the contract (weekly distribution of rewards).

The DAO smart contract is still unknown, so it is also unknown if it will include time-out mechanisms to protect against his kind of scenario, so I must mention it here at least.

Severity classification

- Likelihood: medium, as it is not infrequent that a DAO does not reach consensus
- Impact: high, as it breaks one of the core invariants (weekly rewards)
- Overall severity: **medium**

Recommendation

Consider using different contract roles for different functions. For example:

- `DAO_ROLE`: can change the `_antibottime` or other contract settings
- `CONTRACT_OPERATOR_ROLE`: can call `startNewEpoch()`, pause the contract, etc.

Fixes review

The team has decided to acknowledge this issue but will implement the necessary fixes to the DAO contract. The DAO will be the owner of this staking contract but will allow a certain wallet (spillway's contract owner) to bypass the voting system to execute `startNewEpoch()` without needing consensus from the DAO.

[M-5] All public functions will revert due to a lack of rewards tokens if the owner withdraws too many reward tokens with `emergencyWithdraw()`

Description

Stakers will be unable to stake new funds, withdraw staked funds, or claim rewards if the contract does not have enough reward tokens in its balance. This situation will not happen due to bad accounting of the contract, but it can happen if the contract owner withdraws too many reward tokens from the contract using `emergencyWithdraw()`. The three public functions will revert until new reward tokens are sent to the contract.

The `claim()` function is the one that will revert when attempting to transfer funds:

```
    require(rewardToken.transfer(msg.sender, rewards), "Reward transfer 
failed");
    totalRewardsClaimed = totalRewardsClaimed+rewards;
```

As the functions `stake()` and `unstake()` also call `claim()` internally, they will also revert.

This scenario is highly probable, as some of the previously mentioned issues will lead to some reward tokens locked in the contract, which will be rescued using `emergencyWithdraw()`.

Severity classification

- Likelihood: low, as the contract owner (or DAO) is expected to act in good faith.
- Impact: high, as users would lose access to their funds and rewards.
- Overall severity: **medium**.

Recommendation

When removing token rewards that have been locked in the contract due to some of the previous issues, calculate very precisely how many reward tokens can be withdrawn. If it ended up happening anyway, just send more reward tokens to the contract to fix the situation.

Low

[L-1] Unbounded for loops can increase the gas cost of `_calculateRewards()` significantly, or in extreme cases, make it revert, which locks user funds and rewards in the contract

Impact

The more times a staker stakes within the last two days of an epoch, the more gas the transaction will cost. The gas cost for claiming rewards will increase proportionally to the number of epochs since the last claim took place.

It is possible (but extremely unlikely) that the gas cost reaches the block limit, and the funds and rewards are locked in the contract, due to a user not being able to call any of the functions that internally call `_calculateRewards()`, which are `stake()`, `unstake()`, and `claim()`.

Description

For-loops are dangerous as the gas consumption grows with the size of the array. In extreme situations, the gas consumption can be as high as the block gas limit, making it revert regardless of how much gas the user is willing to spend in the transaction.

There are two unbounded for loops in the `SpillwayStake` contract:

- An iteration over `stakeHistory` for the antibot time logic
- an iteration over the epochs, to collect rewards from pending all epochs

If a user does not claim for multiple epochs in a row, or if the user stakes a large number of times during a single epoch, it can lead to out-of-gas errors, and in the most critical scenario, hitting the out-of-gas limit.

Severity classification

- Likelihood: low, as it is not likely that the user does not stake for so long, or stake such a large amount of times in a single epoch
- Impact: high, as if the limit is hit, the user cannot perform any of the following actions: `stake()`, `unstake()` and `claim()`. Which will mean that the user funds are permanently locked in the contract.
- Overall severity: **low**.

Mitigation

- For the anti-bot period: limit the number of stakes a user can do in a single epoch
- For the number of epochs: a solution is not trivial with the current implementation

[L-2] Underflow error will cause the view function `calculateRewards()` to revert in certain situations

Description

The view function `calculateRewards()` does not check if `stakedAmount < stake.amount` inside the antibot logic, it can revert for certain sequences of non-deleted stakes when the subtraction underflows because the `stakedAmount` is smaller than a certain `staker.stakeHistory[currentEpoch - 1][i].stakedAmount`.

```
    function calculateRewards(address user) public view returns (uint256 rewards) {
        Staker storage staker = stakers[user];
        ...

        uint128 thresholdTimestamp = uint128(epochData[currentEpoch - 1].epochEnd - _antibottime);
        if (staker.stakeHistory[currentEpoch - 1].length > 0 &&
currentEpoch != 2) {
            for (uint256 i = staker.stakeHistory[currentEpoch - 1].length -
1; i >= 0; i--) {
                ...

@>              stakedAmount = stakedAmount -
staker.stakeHistory[currentEpoch - 1][i].stakedAmount;
```

Recommendation

Include a similar check as the one included in `_calculateRewards()`:

```
    function calculateRewards(address user) public view returns (uint256
rewards) {
        Staker storage staker = stakers[user];
        ...

        uint128 thresholdTimestamp = uint128(epochData[currentEpoch -
1].epochEnd - _antibottime);
        if (staker.stakeHistory[currentEpoch - 1].length > 0 &&
currentEpoch != 2) {
            for (uint256 i = staker.stakeHistory[currentEpoch - 1].length -
1; i >= 0; i--) {
                ...
+               if (staker.stakeHistory[endEpoch][i].stakedAmount >
stakedAmount) {
+                   stakedAmount = 0;
+                   break;
+               }
                stakedAmount = stakedAmount -
staker.stakeHistory[currentEpoch - 1][i].stakedAmount;
```

[L-3] The contract implements `emergencyWithdrawETH()` but there is no `receive()` or `fallback()` functions

Description

A contract cannot receive native ETH without the `receive()` or `fallback()` functions, so the function `emergencyWithdrawETH()` is unnecessary. The only exception to this statement is the `selfDestruct` method, which is probably not the intended way of receiving ether in this particular protocol.

Recommendation

If the intention is to receive ether into this contract, needs to have a `receive()` function.

```
+   receive() external payable {}
```

[L-4] Stakers have no guarantee of receiving rewards until the epoch ends

Impact

When a staker stakes, there is no guarantee of when and how much rewards he will receive. If the contract owner decides to discontinue the rewards program, a user will end up staking tokens for nothing.

The contract owner can decide when to finish the epoch, and how many rewards to distribute, and he can decide to discontinue the rewards program.

Recommendation

Fixing this problem is not trivial, as it would require a redesign of the architecture. Something along the lines of depositing the rewards for a period, so that the staker has a guarantee of how much is being distributed, and chooses if they want to stake or not based on that.

[L-5] Precision loss errors when calculating rewards rate

Description

When calculating the rewards rate, this division ignores the reminder.

```
@>    epochData[currentEpoch].epochRewardRate = (_rewards * 1e18) /
totalStaked;
```

The impact is very low, as the rewards are boosted with a `1e18` factor, so the decimals lost will be minimal

Recommendation

Store the reminder in a buffer variable and add it to the next rewards distribution. This is an illustrative and non-gas-efficient way of doing it:

```
+ uint256 bufferedDecimalLost;
    ...

    function startNewEpoch(uint256 _rewards) external onlyOwner {
      ...

+     epochData[currentEpoch].epochRewardRate = ((bufferedDecimalLost +
_rewards) * 1e18) / totalStaked;
+     bufferedDecimalLost = ((bufferedDecimalLost + _rewards) * 1e18) %
totalStaked;
```

[L-6] Consider using SafeERC20 library from Openzeppelin to interact with ERC20 tokens.

Description

Using the native `ERC20.transfer` and `ERC20.transferFrom` is not recommended, as they cannot handle special ERC20 implementations. Using `safeTransfer` and `safeTransferFrom` is more secure, and almost always recommended.

The impact in this particular case is minimal, as the two ERC20s that the staking contract interacts with are trusted and known contracts. However, USDC is a proxy contract, so its implementation could change in the future.

[L-7] Inconsistent/confusing logic when updating the value of `staker.lastEpoch`

Description

The statement `lastClaimedEpoch = currentEpoch - 1;` is used in multiple places in the different functions. In some cases, it is even set twice for the same transaction, wasting gas.

```
  function stake(uint256 amount) external Active {
      require(amount > 0, "Amount must be greater than 0");
      require(spillToken.transferFrom(msg.sender, address(this), amount),
"Staking transfer failed");
      Staker storage staker = stakers[msg.sender];
      uint256 lastClaimedEpoch = staker.lastClaimedEpoch;

      if (staker.totalStakedAmount == 0) {
@>          lastClaimedEpoch = currentEpoch - 1;
      } else {
@>          claim();
      }
      staker.stakeHistory[currentEpoch].push(StakeHistory(amount,
block.timestamp));
      staker.totalStakedAmount = staker.totalStakedAmount + amount;
      // @audit-info This statement is repeated. Also, set inside claim()
@>    staker.lastClaimedEpoch = currentEpoch - 1;
      totalStaked = totalStaked + amount;

      emit Staked(msg.sender, amount);
```

Recommendation

Only set `staker.lastClaimedEpoch = currentEpoch - 1;` inside the `claim()` function, where it logically belongs, as it sets the information of when was the last time the claim was performed.

Informational / Gas

[I-1] Missing events in state-changing functions

The following functions do not emit events, but change the state of the contract:

- `startInitial()`
- `setPaused()`

- `setAntibotTime()`

[I-2] Storage variables are accessed multiple times, increasing gas costs unnecessarily

Reading state variables from storage is one of the most costly operations. Reading it multiple times within the same function increases the transaction gas unnecessarily.

Recommendation

Consider caching into memory storage variables that are read multiple times. Example:

The storage variable `currentEpoch` is read multiple times during `startNewEpoch()`. The value could be stored in a memory variable `_currentEpoch` and used everywhere to reduce the number of storage reads to 1. Note that `_currentEpoch` should be increased, and the new value stored, replacing the `currentEpoch++` statement.

```
    function startNewEpoch(uint256 _rewards) external onlyOwner {
        require(_rewards > 0, "Rewards must be greater than 0");
        epochData[currentEpoch].epochEnd = block.timestamp;
        if (totalStaked > 0) {
            epochData[currentEpoch].epochRewardRate = (_rewards * 1e18) /
totalStaked;
        } else {
            // @audit-info No need to set this to 0, as it was already
initialized as 0 when the epoch started. Save gas by leaving the default
value
            epochData[currentEpoch].epochRewardRate = 0;
            emit Epoch(
                currentEpoch,
                epochData[currentEpoch].epochStart,
                epochData[currentEpoch].epochEnd,
                epochData[currentEpoch].epochRewardRate
            );
            currentEpoch++;
            epochData[currentEpoch] = EpochData(block.timestamp,
block.timestamp, 0);
            return;
        }
        emit Epoch(
            currentEpoch,
            epochData[currentEpoch].epochStart,
            epochData[currentEpoch].epochEnd,
            epochData[currentEpoch].epochRewardRate
        );
        currentEpoch++;
        require(rewardToken.transferFrom(msg.sender, address(this),
_rewards), "Reward transfer failed"); // @audit-info consider using
OpenZeppelin's SafeTransfer for ERC20s
        epochData[currentEpoch] = EpochData(block.timestamp,
block.timestamp, 0);
    }
```

[I-3] Unnecessary initializations in the constructor

```
        // @audit-info Not necessary to initialize the variable, as the
default value is 0.
        currentEpoch = 0;Save gas.
        // @audit-info epochData initialization is useless, as in
`startInitial()` is initialized with the same values again, for the same
value of `currentEpoch`
        epochData[currentEpoch] = EpochData(block.timestamp,
block.timestamp, 0);
```

[I-4] Cache array length to save gas

Description

If the limit of a for-loop is the length of an array, (like in the example below from `_calculateRewards()`), the length of the storage array is read in every iteration of the loop, increasing the gas cost significantly.

```
@>  for (uint256 i = staker.stakeHistory[endEpoch].length - 1; i >= 0; i--)
{
        if (staker.stakeHistory[endEpoch][i].timeofStaking <
thresholdTimestamp) {
            break;
        }
```

Recommendation

Cache the length of the array, and use that as a conditional inside the array:

```diff
+    uint256 length = staker.stakeHistory[endEpoch].length;
-    for (uint256 i = staker.stakeHistory[endEpoch].length - 1; i >= 0; i-
-) {
+    for (uint256 i = length - 1; i >= 0; i--) {
        if (staker.stakeHistory[endEpoch][i].timeofStaking <
thresholdTimestamp) {
            break;
        }
```

[I-5] Logical path inside `_calculateRewards()` is never reached, thus can be removed

Description

The path `staker.lastClaimedEpoch` can never be 0 when the `_calculateRewards()` is called, so the `if` statement in the code snippet below is useless.

```
  function _calculateRewards(address user, uint256 startEpoch, uint256
endEpoch) private returns (uint256 rewards) {
      Staker storage staker = stakers[user];
      rewards = 0;
      //i The lastClaimedEpoch can never be 0 when reaching this point. It
is always pre-set before calling claim()
@>    if (staker.lastClaimedEpoch != 0) {
@>        startEpoch = startEpoch + 1;
@>    }
      // @audit-info the alternative path (lastclaimedEpoch==0) can never
be reached
```

Recommendation

```
  function _calculateRewards(address user, uint256 startEpoch, uint256
endEpoch) private returns (uint256 rewards) {
      Staker storage staker = stakers[user];
      rewards = 0;
-     if (staker.lastClaimedEpoch != 0) {
-         startEpoch = startEpoch + 1;
-     }
      // @audit-info the alternative path (lastclaimedEpoch==0) can never
be reached
```

[I-6] Unnecessary `break` inside for-loop

Inside `_calculateRewards()` the for loop below iterates from `length` to `i >= 0`, and then there is a check if `i==0` to `break` out of the loop. This is unnecessary, as the loop could be simply defined from `length` to `i > 0`, achieving the same result.

```
    for (uint256 i = staker.stakeHistory[endEpoch].length - 1; i >= 0; i--)
{
      ...
      if (i == 0) {
          break; // @audit-info unnecessary statement. Simply change the
for-loop to continue until i > 0
      }
```

This also happens in another for loop, inside another function.

[I-7] Unnecessary timestamp check, as `epochEndTime` can never be higher than `block.timestamp`

The only place where the storage variable `epochData[epoch].epochEnd` is set, it is set to `block.timestamp`, inside `startNewEpoch()`. However, inside `_calculateRewards()`, the following check is in place:

```
@>    if (block.timestamp >= epochEndTime) {
          rewards = rewards + ((epochRewardRate * stakedAmount) / 1e18);
      }
```

This check is unnecessary, as the condition `block.timestamp >= epochEndTime` will **always** be met.

Recommendation

Remove that if-conditional, and save a tiny bit of gas.

```
-     if (block.timestamp >= epochEndTime) {
-         rewards = rewards + ((epochRewardRate * stakedAmount) / 1e18);
-     }
+     rewards = rewards + ((epochRewardRate * stakedAmount) / 1e18);
```

Appendix

BaseForProofOfConcepts

▶ Details

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "lib/forge-std/src/Test.sol";
import "lib/forge-std/src/StdInvariant.sol";
import {SpillwaysStake} from "src/staking.sol";
import {MockERC20} from "test/mock.sol";
import {Handler} from "test/invariant/handler.sol";

contract BaseForProofOfConcepts is Test {
    MockERC20 public spilltoken;
    MockERC20 public usdc;
    SpillwaysStake public staking;

    address user = makeAddr("user");
    address attacker = makeAddr("attacker");
    address owner = makeAddr("owner");

    function setUp() public virtual {
        vm.warp(1672617600 + 7 * 100 days);
        usdc = new MockERC20("USDC", "USDC");
        spilltoken = new MockERC20("SpillToken", "SPILL");

        deal(address(usdc), owner, 1000000 ether);
        vm.startPrank(owner);
        staking = new SpillwaysStake(address(spilltoken), address(usdc));
        staking.startInitial();
        staking.startNewEpoch(10 gwei);
        usdc.approve(address(staking), type(uint256).max);
        vm.stopPrank();
    }
}
```