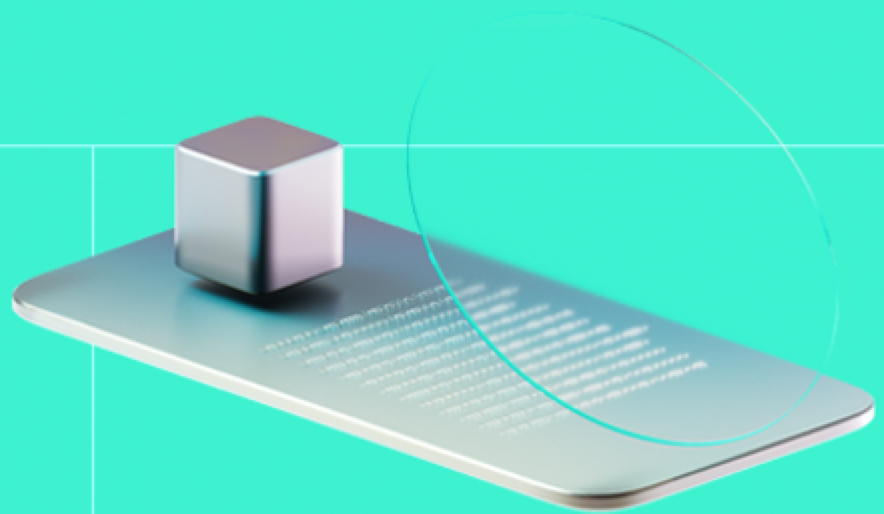
Smart Contract Code Review And Security Analysis Report



Customer: SDAO

Date: 17/06/2024

We express our gratitude to the SDAO team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

SDAO is a staking platform that allows users to earn rewards based on the staked ERC20 token deposit amount and the lock duration.

Platform: EVM

Language: Solidity

Tags: Staking, ERC20

Timeline: 15/04/2024 - 17/06/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/Singularity-DAO/staking-reward-contracts
Commit	827be52

Audit Summary

10/10	10/10	96.72%	8/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 9.7/10

The system users should acknowledge all the risks summed up in the risks section of the report

3	3	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Severity	Count
Critical	1
High	0
Medium	1
Low	1

Vulnerability	Status
F-2024-1335 - Unlock Date Is Not Reset When The Entire Deposit Is Withdrawn	Fixed
F-2024-1336 - Miscalculated deltaScore Allows Malicious Users Earning Rewards For Free	Fixed
F-2024-1343 - Return Values Of transfer()/transferFrom() Not Checked	Fixed

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for SDAO
Audited By	Seher Saylik
Approved By	Ataberk Yavuzer, Kaan Caglan
Website	http://singularitydao.ai/
Changelog - Preliminary Report	18/04/2024
Changelog - Final Report	06/05/2024

Table of Contents

System Overview **6**
Privileged Roles 6
Executive Summary **7**
Documentation Quality 7
Code Quality 7
Test Coverage 7
Security Score 7
Summary 7
Risks **8**
Findings **9**
Vulnerability Details 9
Observation Details 17
Disclaimers 28
Appendix 1. Severity Definitions **29**
Appendix 2. Scope **30**

System Overview

SDAO is a staking protocol with the following contracts:

SDAOLinearSimpleReward — a reward management contract that allows the addition of rewards with an emission period, calculates claimable rewards for the users, and facilitates the claiming process. Additionally, it tracks user shares and reserves pending rewards for users to be claimed later. The owner of the protocol adds the rewards to the system and determines the emission durations.

In this system, the reward ratio is calculated based on the staked duration and the amount deposited. Specifically, the reward is directly proportional to the product of the deposited amount and the stake duration. This means that users who stake a larger amount for a longer period will earn a higher proportion.

The formula used for calculating the reward ratio is typically something like:

- Reward Ratio=Deposited Amount×Stake Duration

Likewise, when withdrawing deposited amounts, the new score is calculated proportionally to the withdrawn amount and the total deposited amount, regardless of how much each deposit was locked. So, the new score is determined by the ratio of the remaining amount to the total deposited amount of the current score.

SDAOLockedStaking — the main staking contract that facilitates the staking of tokens with specified locking periods. Users can deposit tokens and extend their locking periods to increase their score. Withdrawals are allowed after the tokens unlock or immediately with an early unlock fee deducted.
Important points include a maximum locking period of 1000 days, an early unlock fee capped at 50% of the deposited amount when users want to withdraw before the unlock date.

Clonable — a contract that provides functionality for creating and managing clones. It allows the creation of clones with a specified owner, enables ownership transfer, and ensures that only the owner can execute certain functions.

Privileged roles

- The owner of the SDAOLinearSimpleReward contract can initialize the contract, add rewards, extend the reward duration, recover unsupported tokens,
- The owner of SDAOLockedStaking contract can initialoize the contract, enable/disable new deposits, set early unlock fee per day, set zapper contract, recover unsupported tokens, withdraw the collected fees.
- The owner of Clonable contract can set owner after cloning, transfer ownership.

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation Quality score is **8** out of **10**.

- Functional requirements are partially provided.
- Technical description is provided.
- NatSpec is provided but, could be improved.

Code quality

The total Code Quality score is **10** out of **10**.

- The development environment is configured.

Test coverage

Code coverage of the project is **96.72%**(branch coverage).

- Deployment and basic user interactions are covered with tests.
- Interactions by several users and some important scenarios are not tested thoroughly.

Security score

Upon auditing, the code was found to contain **1** critical, **0** high, **1** medium, and **1** low severity issues. All identified issues have been addressed by the SDAO team, resulting in a final security score of **10** out of **10**.

All identified issues are detailed in the "Findings" section of this report.

Summary

The comprehensive audit of the customer's smart contract yields an overall score of **9.7**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- The **platform owner has the authority to extend or shorten the reward emission time,** directly impacting the rewards that have been earned but not yet claimed. In such instances, users will receive the same amount of reward over an extended period of time.
- **Coarse-grained Authorization Model Risks:** The broad authorization model increases the risk of protocol control loss if any authorized address is compromised, potentially leading to unauthorized actions and significant financial loss.
- **Single Entity Upgrade Authority:** The token ecosystem grants a single entity the authority to implement upgrades or changes. This centralization of power risks unilateral decisions that may not align with the community or stakeholders' interests, undermining trust and security.

Findings

Vulnerability Details

F-2024-1336 - Miscalculated deltaScore Allows Malicious Users Earning Rewards For Free - Critical

Description:

The platform permits users to deposit tokens to earn reward tokens. Users have the flexibility to withdraw their deposited tokens at any time, regardless of whether it has surpassed the unlock date or not. Rewards are determined by a score calculated for each user, derived from the deposited amount multiplied by the staking period. However, a flaw in the `withdraw()` function prevents the user's score from being correctly updated upon withdrawal. This flaw results in users continuing to earn rewards even after they have withdrawn their tokens. A malicious user can deposit a certain amount of tokens and immediately withdraw them, thus initiating the process of earning rewards without actually maintaining a valid deposit for any significant duration.

When users withdraw their deposited tokens, their score needs to be updated by calculating a `deltaScore`. The delta score to be decreased from the users' total score should be based on the users' total lock duration, but it has been calculated based on the time elapsed until the withdrawal.

```
function _withdraw(uint256 _amount, address _user) internal {
UserInfo storage user = userInfo[address(_user)];
require(user.amount >= _amount, "!balance");
require(_amount != 0, "!amount");
uint256 originalUnlockDate = user.unlockDate;
uint256 deltaScore;
// when unlock date has passed
if (originalUnlockDate < block.timestamp) {
// extend unlock date
uint256 extensionPeriod = block.timestamp - originalUnlockDate;
deltaScore = user.amount * extensionPeriod;
totalScore += deltaScore;
user.score += deltaScore;
user.unlockDate = block.timestamp;
}
// apply withdrawal amount
user.amount -= _amount;
uint256 withdrawalAmount = _amount;
deltaScore = _amount * (block.timestamp - user.lockDate);
totalScore -= deltaScore;
user.score -= deltaScore;
SDAOSimpleRewardAPI(rewardsAPI).changeUserShares(_user, user.score);
// when not yet completely unlocked, apply early unlock fee
if (user.unlockDate > block.timestamp) {
uint256 earlyUnlockFee = withdrawalAmount * (originalUnlockDate - bl
ock.timestamp) * earlyUnlockFeePerDay
/ 1 days / MAX_PERCENTAGE;
earlyUnlockFees += earlyUnlockFee;
withdrawalAmount -= earlyUnlockFee;
emit PaidEarlyUnlockFee(_user, earlyUnlockFee, originalUnlockDate -
block.timestamp);
}
```

```
IERC20(depositToken).transfer(address(_user), withdrawalAmount);
}
```

Assets:

- SDAOLockedStaking.sol [https://github.com/Singularity-DAO/staking-reward-contracts]

Status: Fixed

Classification

Impact: 5/5

Likelihood: 5/5

Exploitability: Independent

Complexity: Simple

Likelihood [1-5]: 5
Impact [1-5]: 5
Exploitability [0-2]: 0
Complexity [0-2]: 1
Final Score: 4.8 (Critical)
Hacken Calculator Version: 0.6

Severity: Critical

Recommendations

Remediation: Calculate the `deltaScore` based on the total locked period of a user.

Resolution: The SDAO team corrected the delta score calculation in the _withdraw function as follows:

```
deltaScore = user.score * withdrawalAmount / user.amount;
```

The new score calculation is made based on the amount. (Revised commit: 8b12377)

Evidences

PoC:

Reproduce:

Steps to reproduce the issue:

Initialization of the platform

- The platform owner initializes the contract and enables the deposits after adding rewards.
  1.000.000 reward tokens are added for a 2-month emission period.

Execution of deposits

- UserA deposits 1000 tokens for a one-month locking period.
- UserB deposits 1000 tokens for a one-month locking period but, UserB withdraws it immediately right after the deposit.

Claim

- Time advances to 2 months later and both UserA and userB claim their rewards,
  UserA earned reward = 500.000 reward tokens
  UserB earned reward = 500.000 reward tokens
  The reward amount they got is the same although the userB didn't have any valid stake during this 2-month period.

Results:

PoC test script

First, set the variables to:

```
let depositTokensToMint = ethers.utils.parseEther("1000000");
let rewardTokensToMint = ethers.utils.parseEther("1000000");
```

```
const { expect } = require("chai");
const { waffle, ethers, network } = require('hardhat');
const { provider, loadFixture } = waffle;
const { deposit } = require("./fixtures");
const {
tx_options,
now,
depositTokensToMint,
rewardTokensToMint,
ONE_MINUTE,
ONE_HOUR,
ONE_DAY,
ONE_MONTH,
ONE_YEAR
} = require("./utils");


const ERC20 = require('../node_modules/@openzeppelin/contracts/build
/contracts/IERC20Metadata.json');
const { time } = require("@openzeppelin/test-helpers");


// vars


const nothing = 0;
const zero_address = ethers.constants.AddressZero;


const MAX_PERCENTAGE = 10000; // 100.00%
const START_NOW = 0;



describe("SDAOLockedStaking contract", function () {
let lockingPoolsImplementation;
let simpleRewardImplementation;
```

```
let depositToken;
let rewardToken;
let rewardsAPI;
let lockingPools;
let referenceTimestamp;
let addedRewardsTimestamp;
let alreadyClaimed = ethers.utils.parseUnits("0", 18);


before(async () => {
await network.provider.request({ method: "hardhat_reset", params: []
});
[deployer, user, user2, otheruser, zapper] = await provider.getWalle
ts();


sDAOLockedStaking_CF = await ethers.getContractFactory("SDAOLockedSt
aking");
rewardsAPI_CF = await ethers.getContractFactory("SDAOLinearSimpleRew
ard");
testToken_CF = await ethers.getContractFactory("TestToken");

referenceTimestamp = await now();
});


before("Deploy and mint deposit and reward tokens", async () => {
depositToken = await testToken_CF.deploy("DepositToken", "DT", tx_op
tions);
await depositToken.connect(deployer).mint(user.address, depositToken
sToMint, tx_options);
await depositToken.connect(deployer).mint(user2.address, depositToke
nsToMint, tx_options);




rewardToken = await testToken_
```

See more

Files:

```
User reward balance after claims:  500000096450691705751923
User2 reward balance after claims:  499999903549308294248075
User deposit token balance after withdraw:  985000011574074074074075
User2 deposit token balance after withdraw:  1000000000000000000000000
      ✓ Should distribute the rewards based on the time passed (121ms)
```

F-2024-1335 - Unlock Date Is Not Reset When The Entire Deposit Is Withdrawn - Medium

Description:

The current implementation of the SDAOLockedStaking contract exhibits a flaw related to the management of user unlock dates upon withdrawal and subsequent redeposit. When a user withdraws their entire deposited amount by paying the early unlock fee, the variable user.unlockDate is not reset within the withdraw() function. Consequently, when a user attempts to redeposit a new amount of tokens, the new unlock date must surpass the previous unlock date. Failure to meet this condition prevents the user from depositing again, even if their current deposit amount is zero.

```
function _withdraw(uint256 _amount, address _user) internal {
UserInfo storage user = userInfo[address(_user)];
require(user.amount >= _amount, "!balance");
require(_amount != 0, "!amount");
uint256 originalUnlockDate = user.unlockDate;
uint256 deltaScore;
// when unlock date has passed
if (originalUnlockDate < block.timestamp) {
// extend unlock date
uint256 extensionPeriod = block.timestamp - originalUnlockDate;
deltaScore = user.amount * extensionPeriod;
totalScore += deltaScore;
user.score += deltaScore;
user.unlockDate = block.timestamp;
}
// apply withdrawal amount
user.amount -= _amount;
uint256 withdrawalAmount = _amount;
deltaScore = _amount * (block.timestamp - user.lockDate);
totalScore -= deltaScore;
user.score -= deltaScore;
SDAOSimpleRewardAPI(rewardsAPI).changeUserShares(_user, user.score);
// when not yet completely unlocked, apply early unlock fee
if (user.unlockDate > block.timestamp) {
uint256 earlyUnlockFee = withdrawalAmount * (originalUnlockDate - bl
ock.timestamp) * earlyUnlockFeePerDay
/ 1 days / MAX_PERCENTAGE;
earlyUnlockFees += earlyUnlockFee;
withdrawalAmount -= earlyUnlockFee;
emit PaidEarlyUnlockFee(_user, earlyUnlockFee, originalUnlockDate -
block.timestamp);
}
IERC20(depositToken).transfer(address(_user), withdrawalAmount);
}
```

Requirement in the deposit() function that prevents users depositing with an arbitrary unlock time regardless of the previous unlock date.

```
function _deposit(uint256 _amount,
address _depositor,
address _recipient,
uint256 _lockingPeriod) internal returns (uint256 tokensDeposited) {
...
uint256 newEndPeriod = block.timestamp + _lockingPeriod;
require(newEndPeriod >= user.unlockDate, "!unlockDate");
...
```

Assets:

- SDAOLockedStaking.sol [https://github.com/Singularity-DAO/staking-reward-contracts]

Status: Fixed

Classification

Impact: 4/5

Likelihood: 4/5

Exploitability: Independent

Complexity: Simple

Likelihood [1-5]: 4
Impact [1-5]: 4
Exploitability [0-2]: 1
Complexity [0-2]: 1
Final Score: 2.9 (Medium)
Hacken Calculator Version: 0.6

Severity: Medium

Recommendations

Remediation: Reset the `unlockDate` value of a user when the entire deposited amount is withdrawn in the `_withdraw()` function.

Resolution: The SDAO team implemented an if statement in the `_withdraw()` function that resets the unlock date when the entire deposit is withdrawn. (Revised commit: e4a6ad9)

F-2024-1343 - Return Values Of transfer()/transferFrom() Not Checked - Low

Description: Not all **ERC20** implementations `revert()` when there's a failure in `transfer()` or `transferFrom()`. The function signature has a boolean return value and they indicate errors that way instead. By not checking the return value, operations that should have marked as failed, may potentially go through without actually transfer anything.

Affected lines:

./contracts/SDAOLockedStaking.sol

```
148: IERC20(_token).transfer(to, amount);
157: IERC20(depositToken).transfer(msg.sender, fees);
178: IERC20(depositToken).transferFrom(address(_depositor), address(
this), _amount);
232: IERC20(depositToken).transfer(address(_user), withdrawalAmount)
;
```

./contracts/rewards/SDAOLinearSimpleReward.sol

```
61: IERC20(_token).transfer(_user, _amount);
123: IERC20(_token).transferFrom(msg.sender, address(this), _totalAm
ount);
132: IERC20(_token).transfer(to, amount);
```

Assets:

- SDAOLockedStaking.sol [https://github.com/Singularity-DAO/staking-reward-contracts]
- rewards/SDAOLinearSimpleReward.sol [https://github.com/Singularity-DAO/staking-reward-contracts]

Status: Fixed

Classification

Impact: 4/5

Likelihood: 1/5

Exploitability: Independent

Complexity: Simple

Likelihood [1-5]: 1
Impact [1-5]: 4
Exploitability [0-2]: 0
Complexity [0-2]: 1
Final Score: 1.8 (Low)
Hacken Calculator Version: 0.6

Severity: Low

Recommendations

Remediation: To ensure the reliability and security of token transfers in your smart contract, it's crucial to check the return values of the `transfer()` and `transferFrom()` functions. These functions often return a boolean value indicating the success or failure of the transfer operation. By checking this return value, you can accurately determine whether the transfer was successful and handle any potential errors or failures accordingly. Failing to check the return value may lead to unintended and unhandled transfer failures, which could have security and usability implications.

OpenZeppelin's SafeERC20 library can be used to ensure transfers' safety.

Resolution: The SDAO team introduced the SafeERC20 library for all the contracts. (Revised commit: 4c9eb26)

Observation Details

F-2024-1344 - Missing Checks For address(0) When Updating State Variables - Info

Description: In Solidity, the Ethereum address `0x0000000000000000000000000000000000000000` is known as the "zero address". This address has significance because it's the default value for uninitialized address variables and is often used to represent an invalid or non-existent address. The " Missing zero address control" issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior. For instance, a contract might allow tokens to be sent to the zero address without any checks, which essentially burns those tokens as they become irretrievable. While sometimes this is intentional, without proper control or checks, accidental transfers could occur.

Affected code lines:

./contracts/SDAOLockedStaking.sol

```
139: zapperContract = _zapperContract;
```

./contracts/utils/Clonable.sol

```
28: _owner = newOwner;
43: Clonable(newInstance).setOwnerAfterClone(newOwner);
```

Assets:

- SDAOLockedStaking.sol [https://github.com/Singularity-DAO/staking-reward-contracts]
- utils/Clonable.sol [https://github.com/Singularity-DAO/staking-reward-contracts]

Status: Fixed

Classification

Impact: 4/5

Likelihood [1-5]: 2
Impact [1-5]: 4
Exploitability [0-2]: 1
Complexity [0-2]: 1
Final Score: 2.3 (Low)
Hacken Calculator Version: 0.6

Likelihood: 2/5

Recommendations

Remediation: It is strongly recommended to implement checks to prevent the zero address from being set during the initialization of contracts. This can be achieved by adding require statements that ensure address parameters are not the zero address.

Resolution: The SDAO team implemented missing zero checks for the given functions. (Revised commit: 3619112)

F-2024-1345 - Unnecessary Casting As Variable Is Already Of The Same Type - Info

Description: In Solidity, explicitly casting a variable to a type that it already represents is redundant and can lead to confusion and clutter in the code. This unnecessary casting doesn't typically consume additional gas since Solidity's optimizer often removes such redundant conversions during compilation. However, it does affect code readability and may obscure the actual intent of the code, making it harder for developers to understand and maintain. Ensuring that casting is used only when necessary helps maintain clean, clear, and efficient code.

Affected code:

./contracts/SDAOLockedStaking.sol:

```
54: require(address(depositToken) == address(0), "!reinit"); // Vari
able `depositToken` is converted to `address` from type `address`.

170: UserInfo storage user = userInfo[address(_recipient)]; // Varia
ble `_recipient` is converted to `address` from type `address`.

178: IERC20(depositToken).transferFrom(address(_depositor), address(
this), _amount); // Variable `_depositor` is converted to `address`
from type `address`.

203: UserInfo storage user = userInfo[address(_user)]; // Variable `
_user` is converted to `address` from type `address`.

232: IERC20(depositToken).transfer(address(_user), withdrawalAmount)
; // Variable `_user` is converted to `address` from type `address`.
```

Assets:

- SDAOLockedStaking.sol [https://github.com/Singularity-DAO/staking-reward-contracts]

Status: Fixed

Recommendations

Remediation: Review your Solidity code for instances of unnecessary casting where variables are cast to their own type. Remove these redundant casts to enhance code clarity and maintainability. When writing new code, ensure that casting is only applied when changing a variable's type is genuinely needed. This practice helps in keeping the codebase straightforward and understandable, reducing potential confusion and errors associated with misinterpreting the variable types.

F-2024-1346 - Custom Errors In Solidity For Gas Efficiency - Info

Description:

Starting from Solidity version 0.8.4, the language introduced a feature known as "custom errors". These custom errors provide a way for developers to define more descriptive and semantically meaningful error conditions without relying on string messages. Prior to this version, developers often used the **require** statement with string error messages to handle specific conditions or validations. However, every unique string used as a revert reason consumes gas, making transactions more expensive.

Custom errors, on the other hand, are identified by their name and the types of their parameters only, and they do not have the overhead of string storage. This means that, when using custom errors instead of **require** statements with string messages, the gas consumption can be significantly reduced, leading to more gas-efficient contracts.

Affected code:

./contracts/SDAOLockedStaking.sol:

```
54: require(address(depositToken) == address(0), "!reinit");

55: require(_depositToken != address(0), "!depositToken");

56: require(_rewardsAPI != address(0), "!rewardsAPI");

77: require(msg.sender == zapperContract, "!zapperContract");

131: require(_earlyUnlockFeePerDay <= MAX_EARLY_UNLOCK_FEE_PER_DAY,
"!MAX_EARLY_UNLOCK_FEE_PER_DAY");

146: require(_token != address(0), "!token");

147: require(_token != depositToken, "!depositToken");

168: require(_lockingPeriod <= MAX_LOCKING_PERIOD, "MAX_LOCKING_PERI
OD");

169: require(depositsEnabled, "!depositsEnabled");

171: require(_amount != 0 || user.amount != 0, "!amount");

173: require(newEndPeriod >= user.unlockDate, "!unlockDate");

204: require(user.amount >= _amount, "!balance");

205: require(_amount != 0, "!amount");
```

./contracts/utils/Clonable.sol:

```
18: require(_owner == msg.sender, "ERR_OWNER");
23: require(_owner == address(0), "ERR_REINIT");
```

./contracts/rewards/SDAOLinearSimpleReward.sol:

```
83: require(depositContract == address(0), "!reinit");

84: require(_depositContract != address(0), "!depositContract");

85: require(_rewardToken != address(0), "!rewardToken");
```

```
98: require(_totalAmount != 0, "!amount");

99: require(_secondsInPeriod != 0, "!period");

100: require(IERC20(_token).allowance(msg.sender, address(this)) >=
_totalAmount, "!allowance");

101: require(IERC20(_token).balanceOf(msg.sender) >= _totalAmount, "
!balance");

106: require(reward.endOfEmission < block.timestamp
107: || _startOfEmission <= block.timestamp,
108: "!validEmissionPeriod");

130: require(_token != address(0), "!token");

131: require(_token != reward.rewardToken, "!rewardToken");

171: require(msg.sender == depositContract, "!depositContract");
```

Assets:

- SDAOLockedStaking.sol [https://github.com/Singularity-DAO/staking-reward-contracts]
- rewards/SDAOLinearSimpleReward.sol [https://github.com/Singularity-DAO/staking-reward-contracts]
- utils/Clonable.sol [https://github.com/Singularity-DAO/staking-reward-contracts]

Status: Fixed

Recommendations

Remediation: It is recommended to use custom errors instead of revert strings to reduce gas costs, especially during contract deployment. Custom errors can be defined using the error keyword and can include dynamic information.

F-2024-1348 - State Variables That Are Used Multiple Times In a Function Should Be Cached In Stack Variables - Info

Description:

When performing multiple operations on a state variable in a function, it is recommended to cache it first. Either multiple reads or multiple writes to a state variable can save gas by caching it on the stack. Caching of a state variable replaces each Gwarmaccess (100 gas) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses. Saves 100 gas per instance.

./contracts/SDAOLockedStaking.sol

```
177: uint256 _before = IERC20(depositToken).balanceOf(address(this))
; // State variable `depositToken` is used also on line(s): ['178',
'179'].
182: if (user.amount > 0) { // State variable `user` is used also on
line(s): ['171', '185'].
184: uint256 extensionPeriod = newEndPeriod - user.unlockDate; // St
ate variable `user` is used also on line(s): ['173'].
212: deltaScore = user.amount * extensionPeriod; //State variable `u
ser` is used also on line(s): ['204'].
206: uint256 originalUnlockDate = user.unlockDate; // State variable
`user` is used also on line(s): ['225'].
```

./contracts/rewards/SDAOLinearSimpleReward.sol

```
122: emit UpdatedRewardEmission(reward.totalAmount, reward.startOfEm
ission, reward.endOfEmission); // State variable `reward` is used al
so on line(s): ['113', '104'].
106: require(reward.endOfEmission < block.timestamp // State variabl
e `reward` is used also on line(s): ['112', '111', '112', '122'].
110: uint256 start = (reward.lastClaim != 0) ? reward.lastClaim : re
ward.startOfEmission; // State variable `reward` is used also on lin
e(s): ['110'].
110: uint256 start = (reward.lastClaim != 0) ? reward.lastClaim : re
ward.startOfEmission; // State variable `reward` is used also on lin
e(s): ['120', '122'].
143: uint256 start = (reward.lastClaim != 0) ? reward.lastClaim : st
artOfEmission; // State variable `reward` is used also on line(s): [
'143', '139'].
148: : reward.endOfEmission - start; // State variable `reward` is u
sed also on line(s): ['145', '146'].
157: if (userInfo[_user].shares == 0) return 0; // State variable `u
serInfo` is used also on line(s): ['161'].
164: - userInfo[_user].rewardFloor; // State variable `userInfo` is
used also on line(s): ['157', '161'].
189: if (claimable != 0 && totalShares != 0) { // State variable `to
talShares` is used also on line(s): ['192'].
195: if (userInfo[_user].shares == 0) return; // State variable `use
rInfo` is used also on line(s): ['199', '196'].
```

Assets:

- SDAOLockedStaking.sol [https://github.com/Singularity-DAO/staking-reward-contracts]

Status: Fixed

Recommendations

Remediation: Cache state variables in stack or local memory variables within functions when they are used multiple times. This approach replaces costlier Gwarmaccess operations with cheaper stack reads, saving approximately 100 gas per instance and optimizing overall contract performance.

F-2024-1349 - Redundant State Variable Getters in Solidity - Info

Description:

In Solidity, state variables can have different visibility levels, including `public`. When a state variable is declared as `public`, the Solidity compiler automatically generates a getter function for it. This implicit getter has the same name as the state variable and allows external callers to query the variable's value.

A common oversight is the explicit creation of a function that returns the value of a public state variable. This function essentially duplicates the functionality already provided by the automatically generated getter. For instance, if there's a public state variable `uint256 public value`;, there's no need for a function like `function getValue() public view returns (uint256) { return value; }`, as the compiler already provides a `value()` function.

Affected code:

./contracts/rewards/SDAOLinearSimpleReward.sol:

```
40: function getRewardInfo() external view override returns (RewardT
okenInfo memory) {
41: return reward;
42: }
```

Assets:

- rewards/SDAOLinearSimpleReward.sol [https://github.com/Singularity-DAO/staking-reward-contracts]

Status: Mitigated

Recommendations

Remediation:

Avoid creating explicit getter functions for 'public' state variables in Solidity. The compiler automatically generates getters for such variables, making additional functions redundant. This practice helps reduce contract size, lowers deployment costs, and simplifies maintenance and understanding of the contract.

F-2024-1350 - Functions Not Used Internally Can Be Marked As External - Info

Description: The function `transferOwnership()` is currently set to public visibility but is never called internally. Public functions cost more Gas than external functions.

```
function transferOwnership(address newOwner) public onlyOwner {
_owner = newOwner;
}
```

Assets:

- utils/Clonable.sol [https://github.com/Singularity-DAO/staking-reward-contracts]

Status: Fixed

Recommendations

Remediation: Change the given function's visibility to external.

F-2024-1372 - Constructor and initialize() Can Be Marked As Payable - Info

Description:

payable functions cost less gas to execute, since the compiler does not have to add extra checks to ensure that a payment wasn't provided.

A **constructor** can safely be marked as **payable**, since only the deployer would be able to pass funds, and the project itself would not pass any funds.

`constructor()` function in the `Clonable` contract and `initialize()` function in `SDAOLockedStaking` and `SDAOLinearSimpleReward` contracts are not declared as payable.

Assets:

- SDAOLockedStaking.sol [https://github.com/Singularity-DAO/staking-reward-contracts]
- rewards/SDAOLinearSimpleReward.sol [https://github.com/Singularity-DAO/staking-reward-contracts]
- utils/Clonable.sol [https://github.com/Singularity-DAO/staking-reward-contracts]

Status: Fixed

Recommendations

Remediation: Mark constructors as 'payable' in Solidity contracts to reduce gas costs, as this eliminates the need for the compiler to add checks against incoming payments. This is safe because only the deployer can send funds during contract creation, and typically no funds are sent at this stage.

F-2024-1374 - Missing Event Emitting - Info

Description: Events for critical state changes should be emitted for tracking things off-chain.

`setDepositsEnabled()`, `setEarlyUnlockFeePerDay()`, `setZapperContract()` and `claim()` functions in `SDAOLockedStaking` contract do not emit any event although they make important state updates.

`transferOwnership()` function in Clonable contract does not emit an event.

Assets:

- SDAOLockedStaking.sol [https://github.com/Singularity-DAO/staking-reward-contracts]
- utils/Clonable.sol [https://github.com/Singularity-DAO/staking-reward-contracts]

Status: Fixed

Recommendations

Remediation: Create and emit related events.

Resolution: The SDAO team implemented the required events for the given functions. (Revised commit :3859118)

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details

Repository	https://github.com/Singularity-DAO/staking-reward-contracts
Commit	827be52
Whitepaper	https://github.com/hknio/staking-reward-contracts/blob/main/README.md
Requirements	https://github.com/hknio/staking-reward-contracts/blob/main/README.md
Technical Requirements	https://github.com/hknio/staking-reward-contracts/blob/main/README.md

Contracts in Scope

contracts/rewards/SDAOLinearSimpleReward.sol

contracts/rewards/SDAOSimpleRewardAPI.sol

contracts/utils/Clonable.sol

contracts/SDAOLockedStaking.sol