ARMUR Audit Report

Prepared for Cruize Finance

PRESENTED TO

Prithvi

PRESENTED BY

Akhil Sharma
Amritansh

Mar 06, 2023

Contents

CRUIZE STORAGE.SOL DOCUMENTATION

This is a Solidity smart contract named "CruizeStorage" that contains variables and mappings for a financial application. The contract is licensed under the MIT License and uses Solidity version 0.8.6.

The following is a brief summary of the variables and mappings in the contract:

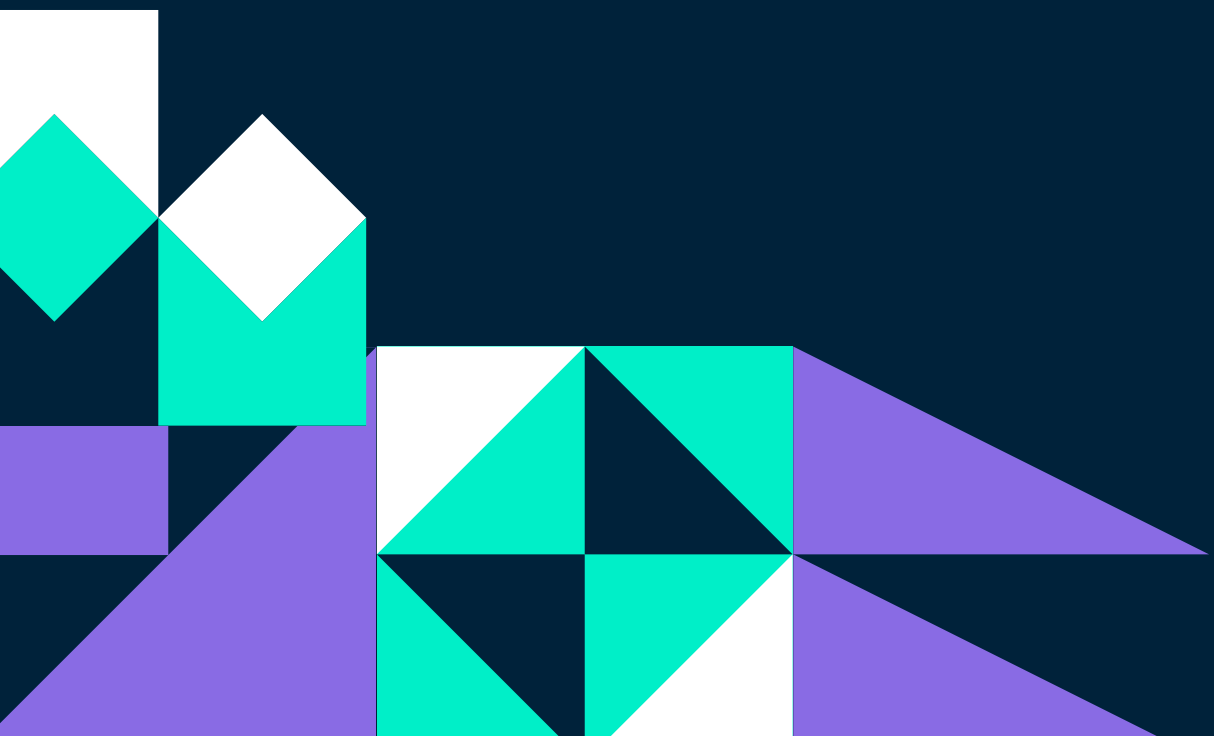
Immutable & Constants:

- feeRecipient: the address that receives management fees
- performanceFee: the fee charged on premiums earned in rollToNextOption. Only charged when there is no loss.
- isPerformanceFeeEnabled: boolean that indicates whether performance fee is enabled
- managementFee: the management fee charged on entire AUM in rollToNextOption. Only charged when there is no loss.
- isManagementFeeEnable: boolean that indicates whether management fee is enabled
- module: the address that is in charge of weekly vault operations such as transfer fund and burn tokens
- gnosisSafe: the address of gnosis safe where all the fund will be locked
- cruizeProxy: the address of gnosis safe where all the fund will be locked
- crContract: the address of the crContract that will be used to create new crcontract clone
- ETH: the address of ETH
- FEE_MULTIPLIER: the multiplier used to calculate fees
- WEEKS_PER_YEAR: the number of weeks per year used to calculate fees

Mappings:

- cruizeTokens: crtokens minted for user's shares
- isDisable: mapping that enables/disables tokens
- vaults: stores vaults state for every round
- lastQueuedWithdrawAmounts: the queued withdrawal amount in the last round
- currentQueuedWithdrawalShares: the queued withdraw shares for the current round
- withdrawals: stores pending user withdrawals
- depositReceipts: stores the user's pending deposit for the round
- roundPricePerShare: stores the pricePerShare value of an crToken token on every round's close
- tokens: an array of token addresses
- Reserved Space:
- __cruize_gap: empty reserved space that allows future versions to add new variables without shifting down storage in the inheritance chain.

Overall, this contract is a storage contract that holds data for a financial application. The mappings and variables are used to keep track of various aspects of the application, including fees, tokens, and user withdrawals and deposits.



CRUIZE STORAGE.SOL

ISSUES

- Typo in the isManagementFeeEnable variable: The variable is intended to indicate whether the management fee is enabled, but it has a typo in its name (isManagementFeeEnable instead of isManagementFeeEnabled). This could cause confusion for anyone using the variable.

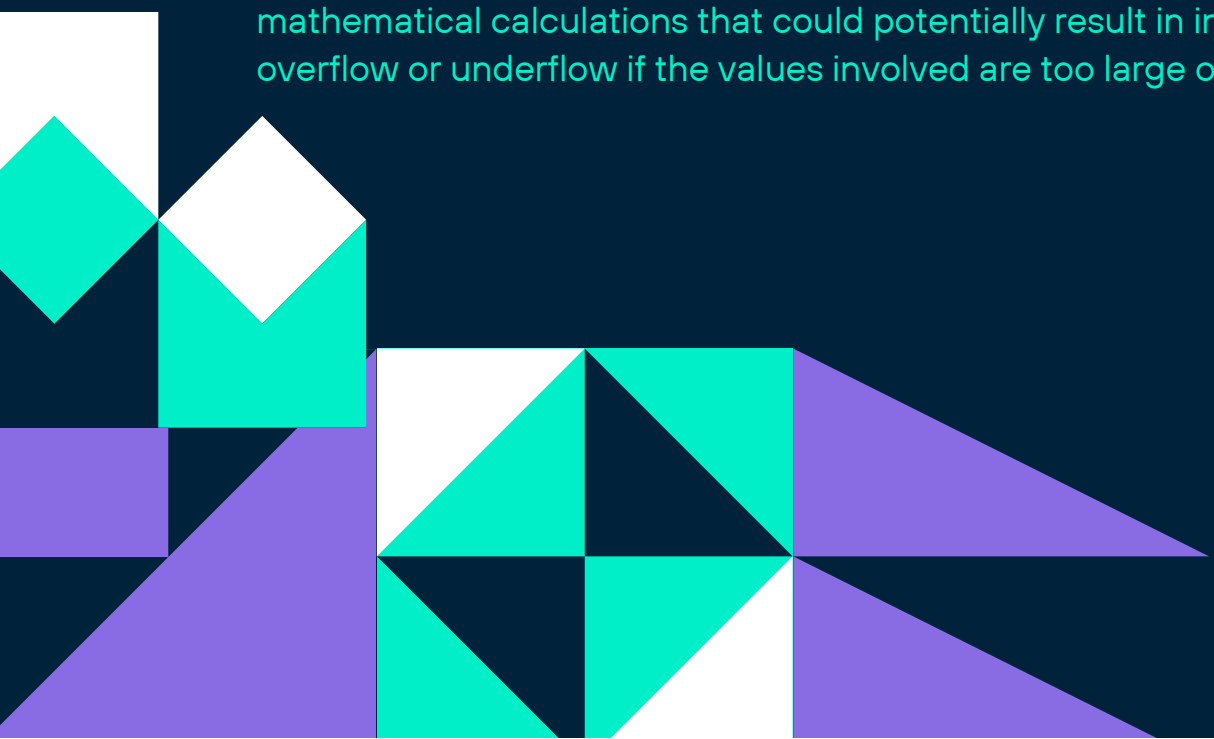
- Lack of access control: The contract doesn't have any access control mechanisms in place to restrict who can perform certain actions. This could potentially allow unauthorized users to manipulate the contract in unintended ways.

- Lack of validation checks: The contract doesn't seem to perform any validation checks on user inputs, such as the amount of funds being deposited or withdrawn. This could lead to errors or exploits if users provide unexpected inputs.

- Lack of documentation: The contract doesn't have sufficient documentation to explain the purpose of each function and variable, which could make it difficult for users to understand and interact with the contract.

- Potential for reentrancy attacks: The contract includes mappings for withdrawals and deposits, which could make it vulnerable to reentrancy attacks if not implemented carefully.

- Potential for integer overflow/underflow: The contract includes some mathematical calculations that could potentially result in integer overflow or underflow if the values involved are too large or too small.



- The "CruizeVault" contract is an abstract contract that acts as a parent contract to several other contracts. It imports and uses multiple other contracts like "Getters.sol", "Helper.sol", "Setters.sol", "ICRERC20.sol", "SharesMath.sol", "Module.sol", "PausableUpgradeable.sol", "ReentrancyGuardUpgradeable.sol", "IERC20.sol", and other OpenZeppelin contracts.

- The contract uses SafeMath and SafeERC20 libraries for arithmetic operations and ERC20 token transfers. The main purpose of this contract is to implement deposit and withdrawal functions for ERC20 tokens. The deposit function updates the deposit info and stores the tokens in the vault. The withdrawal function allows users to withdraw their deposited tokens either instantly or after a period of time.

- The deposit function is implemented in the "_updateDepositInfo" function. The function takes the token address and the amount as arguments. It then updates the deposit receipt for the current user and token, based on the current round, the amount, and the total deposit. It also updates the total pending amount for the token in the vault.

- The "_instantWithdrawal" function allows users to instantly withdraw their deposited tokens. It takes the token address and the amount as arguments. The function checks if the withdrawal round is the same as the current round and if the user has sufficient funds to withdraw. It then transfers the tokens to the user.

- The "_initiateStandardWithdrawal" function allows users to initiate a standard withdrawal, which can be processed after the current round completes. It takes the token address and the number of shares as arguments. The function checks if the user has sufficient unredeemed shares and redeems the shares before initiating the withdrawal. It then stores the withdrawal information in the "withdrawals" mapping.

- The contract also includes functions for redeeming shares, updating the round price per share, and getting the decimals of the token. It is a parent contract to several other contracts that implement specific functionalities like rebasing, staking, and governance.

CRUIZE VAULT.SOL

ISSUES

- In the _updateDepositInfo function, the line ShareMath.assertUint128(newTotalPending) is used to check if newTotalPending is less than or equal to 2^128 - 1. If this condition is not met, the function will throw an error. However, it is not clear why this check is necessary, and it may cause problems if the contract needs to handle larger values.

- The _instantWithdrawal function transfers tokens to the user's account without checking the contract's balance for the given token. If the contract does not have enough funds, the function will fail, and the user's transaction will revert. It would be better to check the contract's balance before attempting the transfer.

- In the _initiateStandardWithdrawal function, there is a possibility that the withdrawals mapping can be written to twice if a user attempts to initiate a withdrawal in a different round before the current withdrawal round is processed. If this happens, the user may lose shares because the shares value in the first withdrawals mapping will be overwritten by the second one.

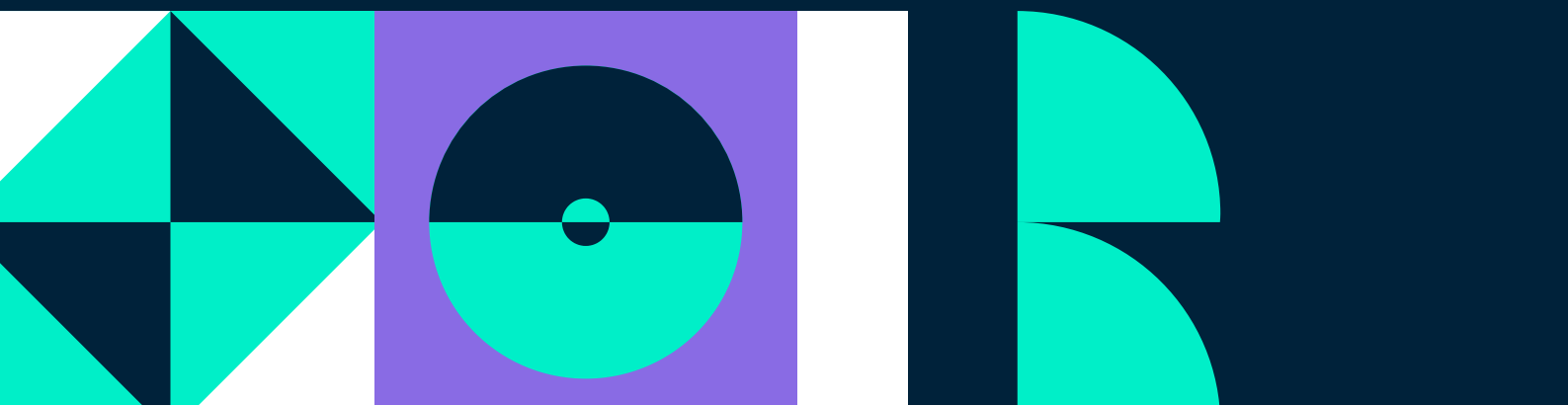
- The contract uses OpenZeppelin's SafeMath and SafeERC20 libraries, which are considered to be secure. However, the SafeCast library is not used consistently throughout the contract, which could result in integer overflow vulnerabilities. It would be best to use SafeCast consistently in the contract.

- The contract is using version 0.8.6 of Solidity, which is not the latest version. Using the latest version of Solidity may help avoid security vulnerabilities and keep the contract up-to-date with the latest improvements in the language.

- It is not clear from the code snippet what the contract is supposed to do or how it interacts with other contracts. Without this information, it is impossible to know whether the code has logical errors or is inefficient.

GETTERS.SOL DOCUMENTATION

- The getters defined inside this contract are as follows:
- getVaultFees: This function calculates the management fee for the current round, based on the asset, total balance, pending amount, and last locked amount.
- shareBalances: This function returns the share balance split between the account and the vault holdings for a given asset. It takes account and token as parameters.
- balanceOfUser: This function returns the user's total locked amount in the strategy for a given asset. It takes account and token as parameters.
- totalBalance: This function returns the vault's total balance, including the amounts locked into a strategy. It takes token as a parameter.
- totalSupply: This function returns the asset total supply. It takes token as a parameter.
- balanceOf: This function returns the cruToken balance. It takes account and token as parameters.



GETTERS.SOL ISSUES

- The smart contract uses the SafeMath library for arithmetic operations, which is a good practice. However, it is important to ensure that all arithmetic operations are properly protected against integer overflows and underflows.

- The shareBalances function returns three values, which could be a gas-intensive operation depending on the size of the values being returned. It might be more efficient to split this function into separate functions that return each value separately.

- The smart contract uses a few uninitialized variables like _performanceFeeInAsset and _managementFeeInAsset. If these variables are not set properly, it could lead to unexpected results.

- The totalBalance function uses the gnosisSafe variable, which is not defined in the code provided. If this variable is not defined or set properly, it could lead to unexpected results.

- setCap(address token, uint256 newCap): Sets a new cap for deposits for a particular token.
- setFeeRecipient(address newFeeRecipient): Sets the new fee recipient address.
- setFeeStatus(bool _performanceFee, bool _managementFee): Sets the management and performance fee status to either enabled or disabled.
- setManagementFee(uint256 newManagementFee): Sets the management fee for the vault.
- setPerformanceFee(uint256 newPerformanceFee): Sets the performance fee for the vault.
- changeAssetStatus(address token, bool status): Enables or disables the asset status, which allows for stopping or resuming token operations.

SETTERS.SOL ISSUES

- Use of external contracts: The smart contract is importing and using functions from other contracts, such as Modifiers.sol, OwnableUpgradeable.sol, SharesMath.sol, and Events.sol. It is important to ensure that these external contracts are trustworthy and well-audited to avoid potential vulnerabilities.

- Lack of access control: The smart contract has some functions that can only be called by the owner, but other functions do not have any access control. This can potentially lead to unauthorized changes to the contract's state.

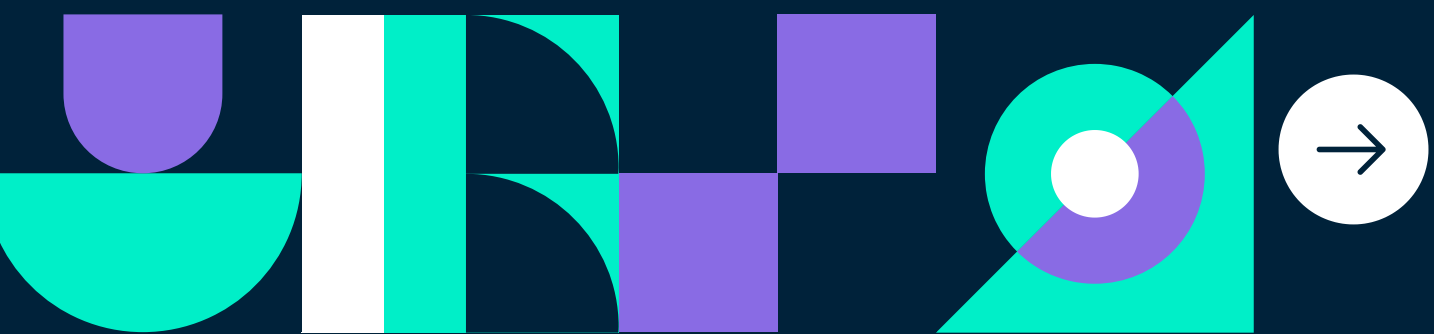
- Lack of input validation: While some functions have input validation, others do not. For example, setFeeRecipient() does not check if the input address is valid or not. It is important to validate all inputs to avoid potential errors or attacks.

- Revert without reason: Some functions revert if the input is invalid, but they do not provide any reason for the revert. This can make it difficult for developers to debug their code and can also confuse users.

- Lack of comments: The smart contract lacks comments explaining the purpose of the functions and the expected inputs and outputs. This can make it difficult for other developers to understand and use the contract.

- Potential overflow: The setManagementFee() function divides the annualized management fee by the number of weeks in a year, which could potentially result in an overflow if the input is too large. It is important to ensure that the input is properly checked and handled to avoid this issue.

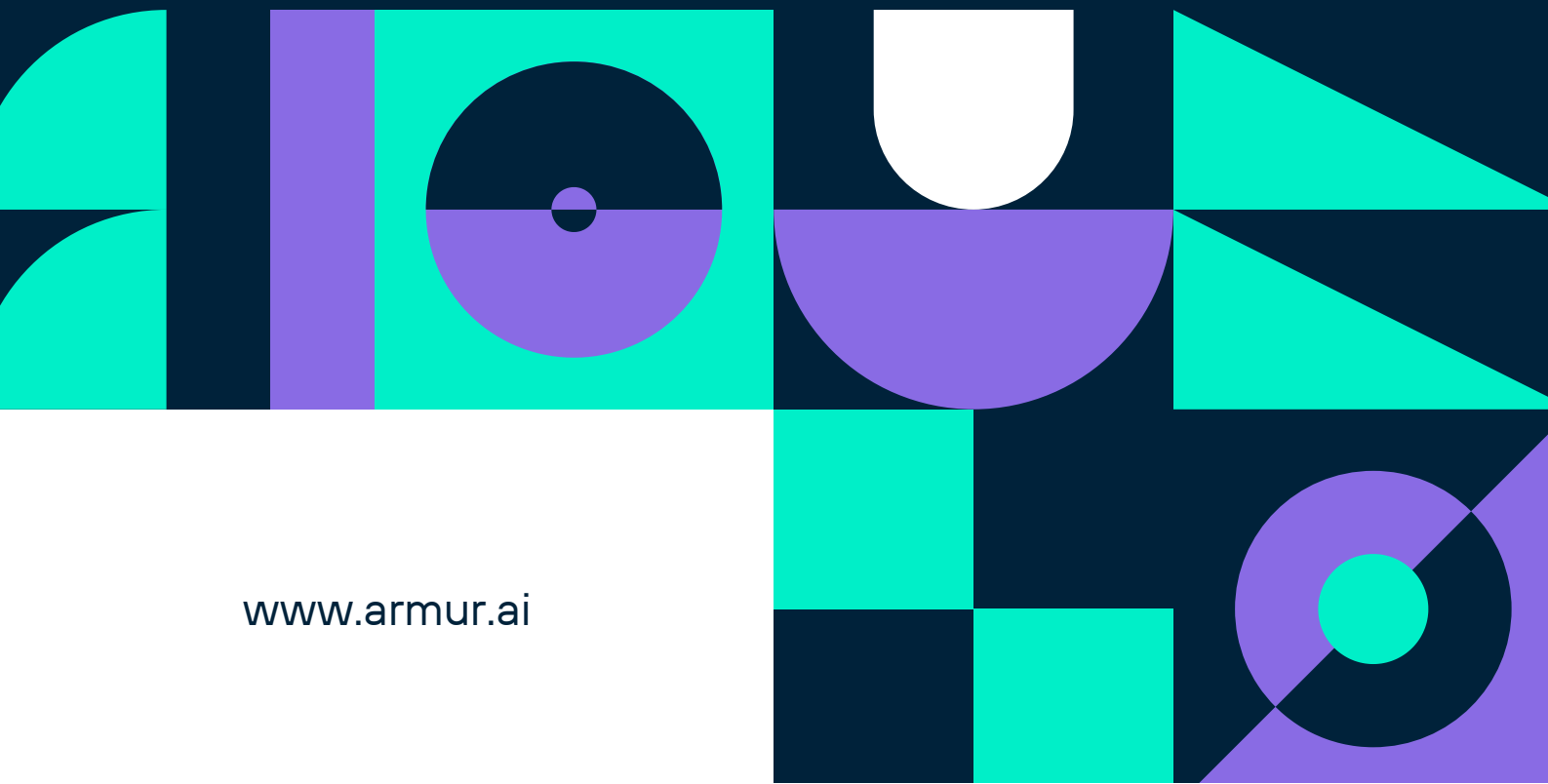
- Potential underflow: The setCap() function uses ShareMath.assertUint104() to check that the input is not too large, but it does not check if the input is smaller than the existing balance in the vault. This can potentially lead to an underflow if the new cap is smaller than the current balance.



CRTOKEN.SOL ISSUES

- It's not clear what the purpose of the _decimal variable is. It's only used in the decimals() function, and could be removed if not needed.
- The initialize() function is marked as external, but could be marked as public to make it clear that it can be called by anyone.
- The contract does not have any function to set the name or symbol of the token. If the token's name and symbol need to be updated, a new contract would need to be deployed.
- The contract imports from OpenZeppelin's ERC20Upgradeable and OwnableUpgradeable contracts, which is a good practice. However, it's important to ensure that the imported versions are up-to-date and do not contain any vulnerabilities.

Thanks.



www.armur.ai