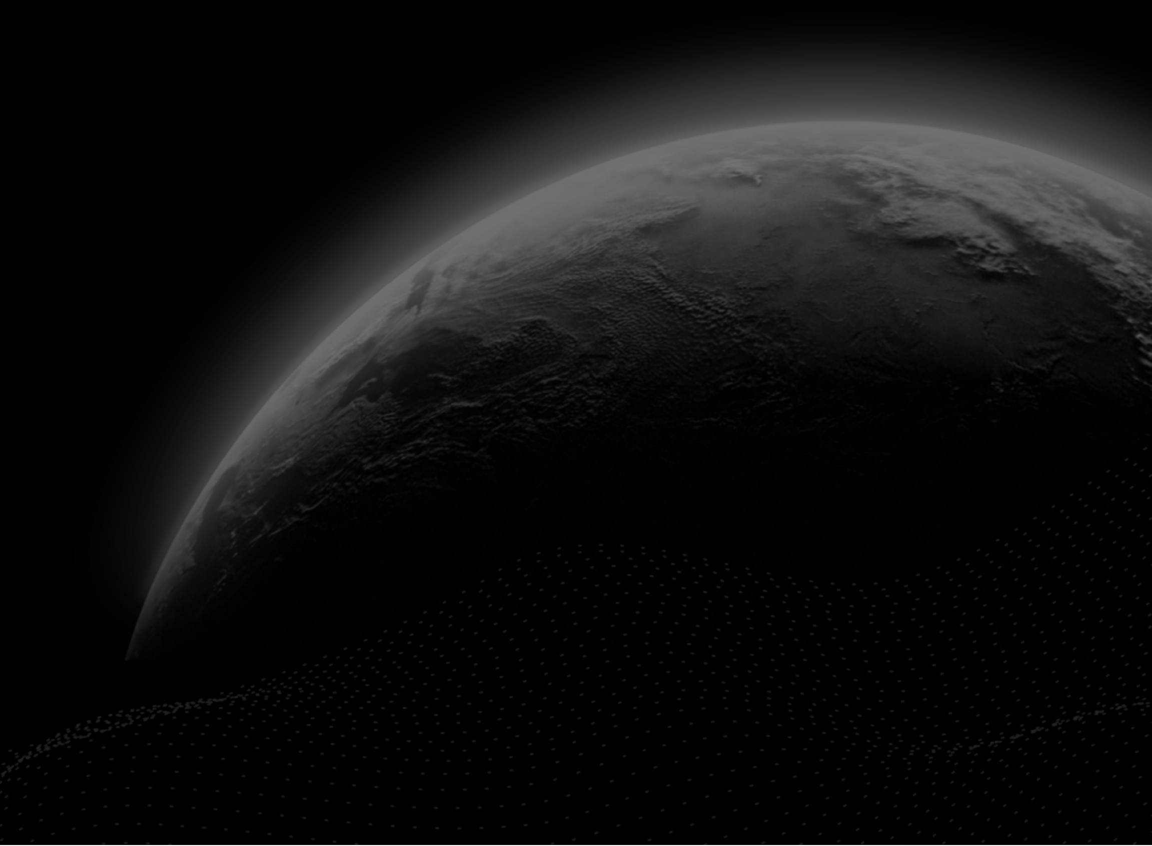
CERTIK

Security Assessment

Golden Goose

CertiK Assessed on Oct 24th, 2024



CertiK Assessed on Oct 24th, 2024

Golden Goose

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES	ECOSYSTEM	METHODS
Staking	Binance Smart Chain (BSC) | Ethereum (ETH)	Formal Verification, Manual Review, Static Analysis

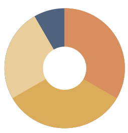
LANGUAGE	TIMELINE	KEY COMPONENTS
Solidity	Delivered on 10/24/2024	N/A

CODEBASE

Private shared.

View All in Codebase Page

Vulnerability Summary



12	8	0	0	4	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined

0	Critical			Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
4	Major	2 Resolved, 2 Acknowledged		Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.
4	Medium	4 Resolved		Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
3	Minor	1 Resolved, 2 Acknowledged		Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
1	Informational	1 Resolved		Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | GOLDEN GOOSE

CODEBASE | GOLDEN GOOSE

Repository

Private shared.

AUDIT SCOPE | GOLDEN GOOSE

6 files audited ● 6 files without findings

ID	Repo	File	SHA256 Checksum
● IDS	CertiKProject/certik-audit-projects	interfaces/IDataStorage.sol	2fccf464cdb77199d8d6b6795e4ee38c2389 1f0aca081a6a3773fb4643492d3f
● DSG	CertiKProject/certik-audit-projects	DataStorage.sol	f5375daaeae8503659474b3d1f3c417bacdd 88e6b673a88021f6d0feb5bdeba1
● LRG	CertiKProject/certik-audit-projects	LRTVault.sol	a85a04fcaa6353cdd3293388c756b5d0595 6b208c8ccf69efc5ccf985a57f29b
● LTG	CertiKProject/certik-audit-projects	LpToken.sol	fd0ac2805b5be277e4f3b8273adcf899bd49 d647fb9a56dbb151a92cd3d78f7c
● USG	CertiKProject/certik-audit-projects	USDVault.sol	05776e488a41be9e7c6ede06baa58fce37b 6557b783692771192e6f810b6d183
● VFG	CertiKProject/certik-audit-projects	VaultFactory.sol	f5abe0559ccd3d138da50f0fcfff641512733e 41789f17171d2834dea6030b6c

APPROACH & METHODS | GOLDEN GOOSE

This report has been prepared for Golden Goose to discover issues and vulnerabilities in the source code of the Golden Goose project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Formal Verification, Manual Review, and Static Analysis techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

REVIEW NOTES | GOLDEN GOOSE

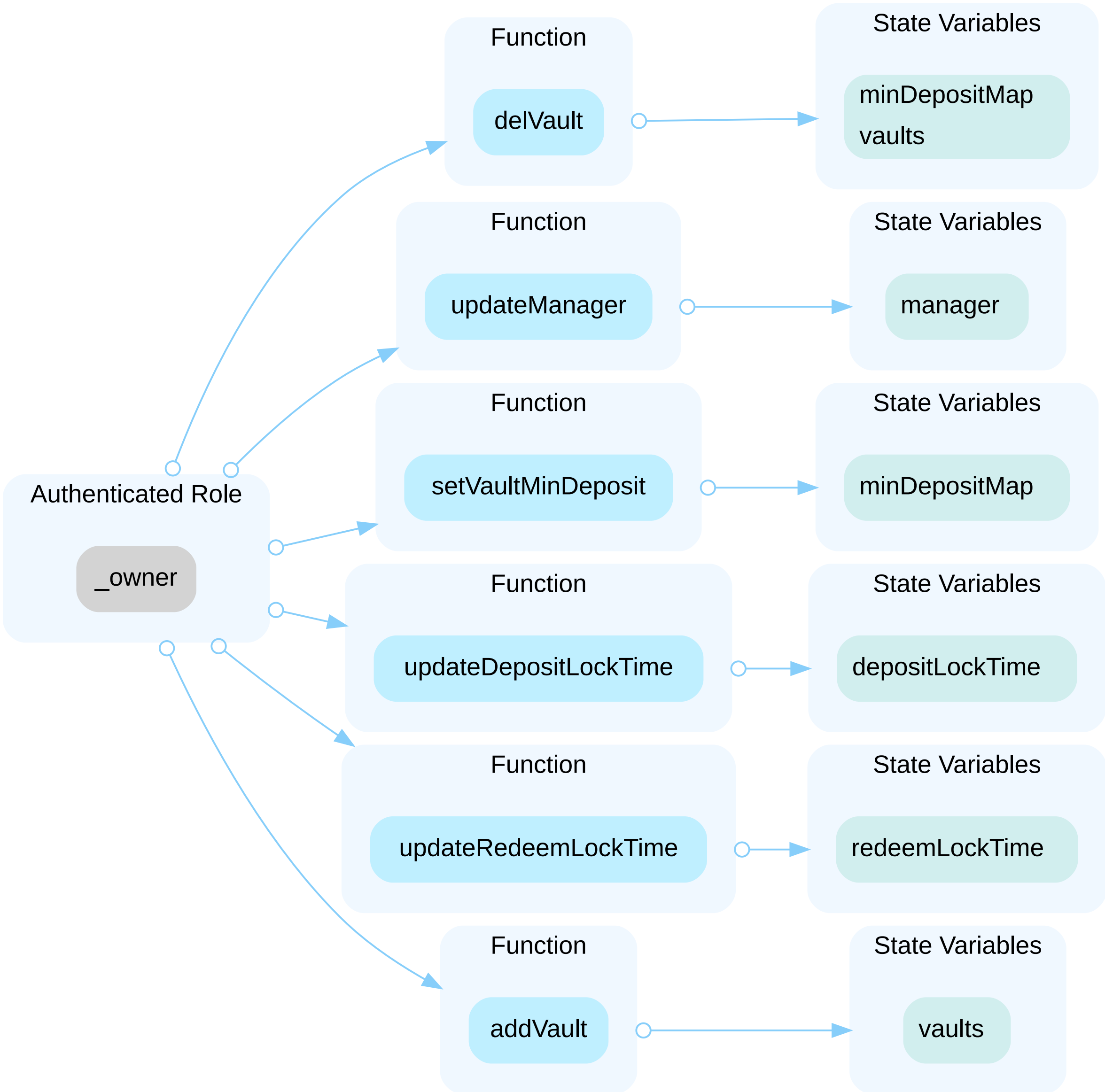
System Overview

The "Golden Goose" project operates as a staking platform and comprises the following contracts:

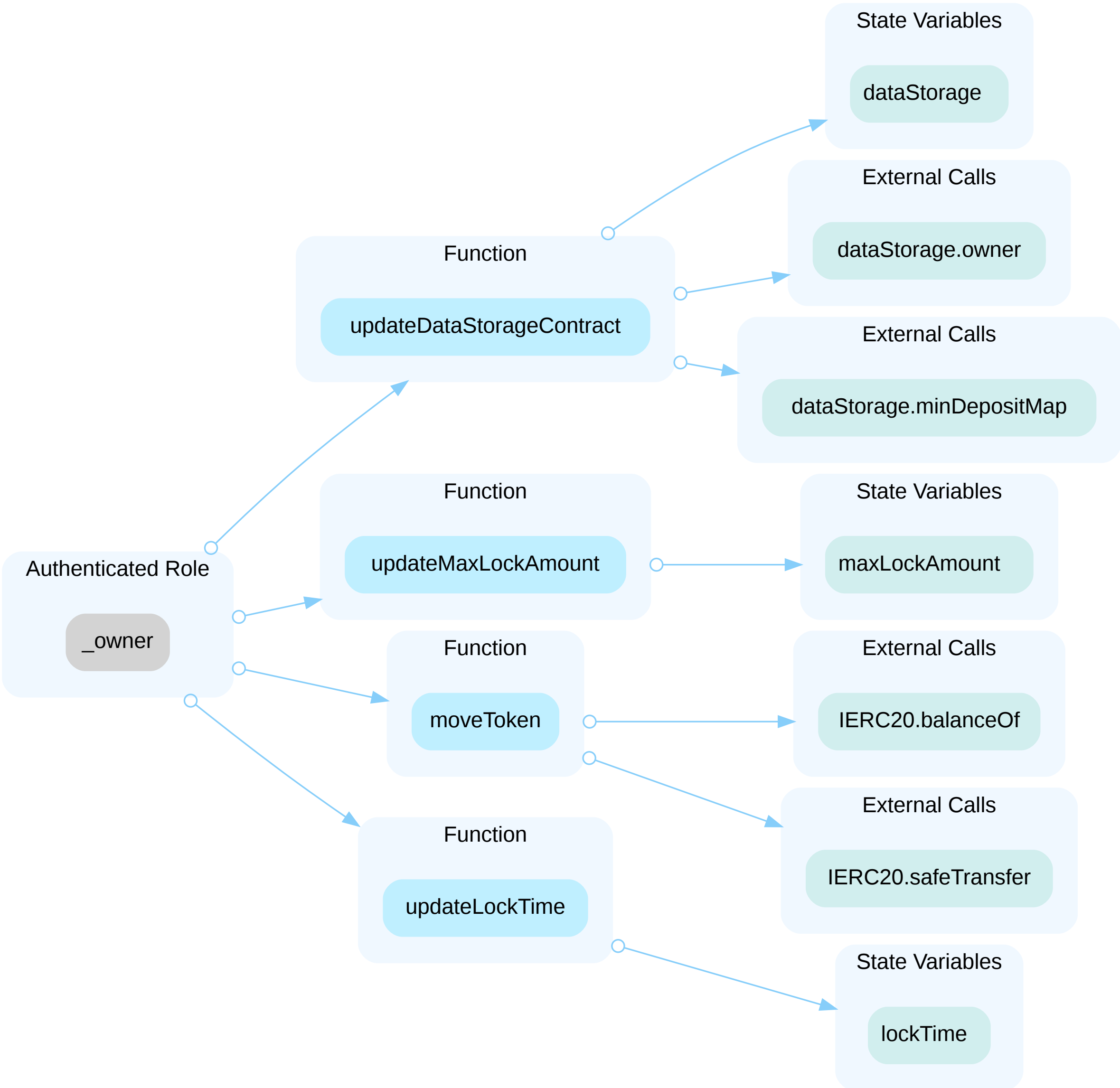
- DataStorage 0x857aB0b4F236F7DD7E5AC5F96C0bbEbF230c2D3B : Records critical variables including `_owner` , `manager` , `depositLockTime` , `redeemLockTime` , `minDepositMap` , and `vaults` for the entire project.
- VaultFactory 0x739A8F9cB6Ec2B79006554dbc3a42fbF75303d18: Creates two types of vaults: `LRTVault` and `USDVault` for users to stake and withdraw tokens. Key differences include:
    - In `LRTVault` , stakers can only withdraw the same amount as the staked amount. In `USDVault` , stakers may withdraw a different amount, controlled by the `_owner` of the `DataStorage` contract via the `LpToken` contract.
    - In `LRTVault` , stakers deposit tokens to the contract. In `USDVault` , deposited tokens are transferred to the `custodian` role.
    - In `LRTVault` , stakers must "unlock" their stakes before withdrawal. In `USDVault` , stakers must "unlock" stakes and "redeem" tokens before withdrawal.
- USDT Vault 0xe8a01d8dac4af19ec7a22cf87f3d141ce6e7e9fb: A `USDVault` that allows staking `USDT` .
- USDT LpToken 0xa79d807b260af533bd481a97039268c028108609: Controls the withdrawable amount in `USDVault` contracts.
- DC_tBTC Vault 0xd31fab00f39153a8389fb9e7065b0c290e1bad5d: An `LRTVault` allowing staking of `DC_tBTC` .
- DC_wstETH Vault 0x234c013dccb6af642fcb7060a91c9c71504f6299: An `LRTVault` allowing staking of `wstETH` .

The system grants the `_owner` control through privileged functions, as detailed in "GLOBAL-01: Centralization Related Risks":

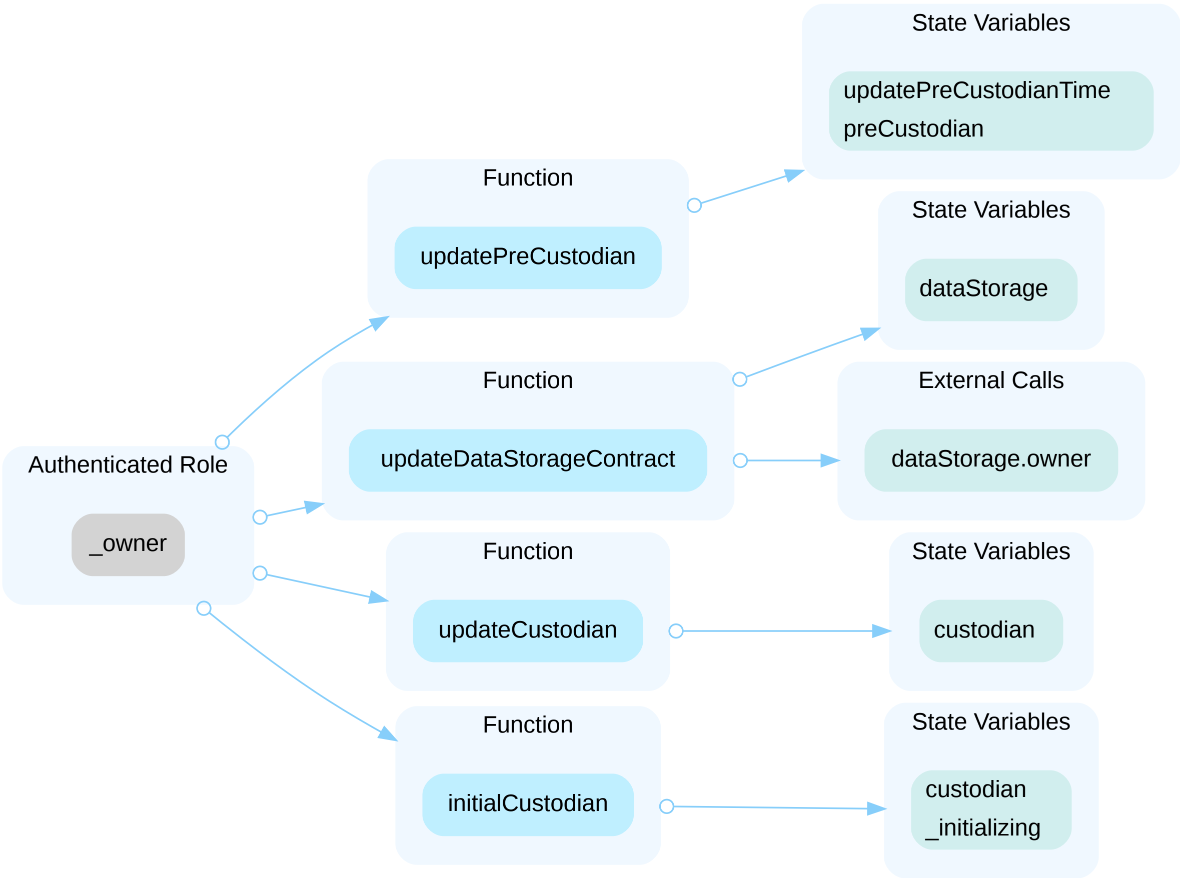
- In the contract `DataStorage` , the role `_owner` has authority over the functions shown in the diagram below:



- In the contract `LRTVault`, the role `_owner` of the `dataStorage` contract has authority over the functions shown in the diagram below:

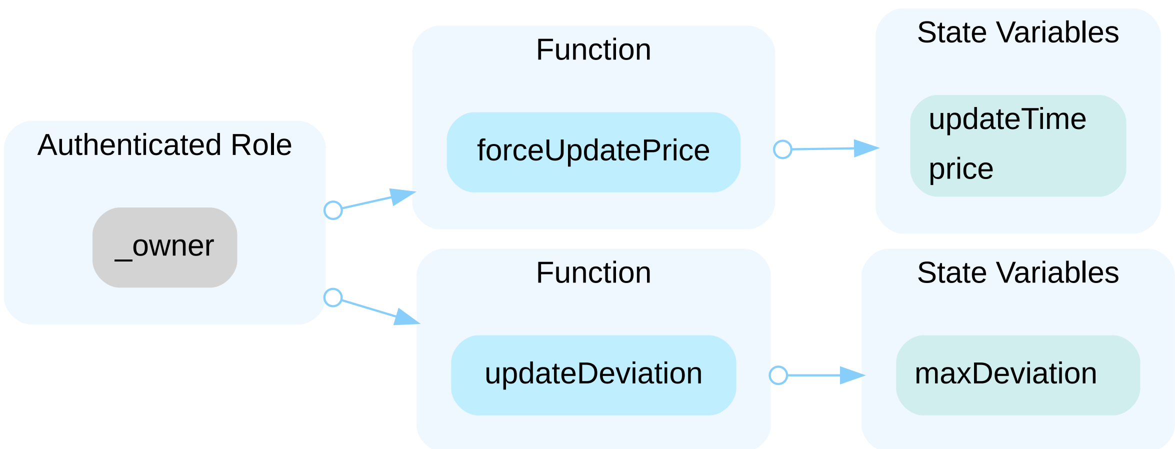


- In the contract `USDVault`, the role `_owner` of the `dataStorage` has authority over the functions shown in the diagram below:

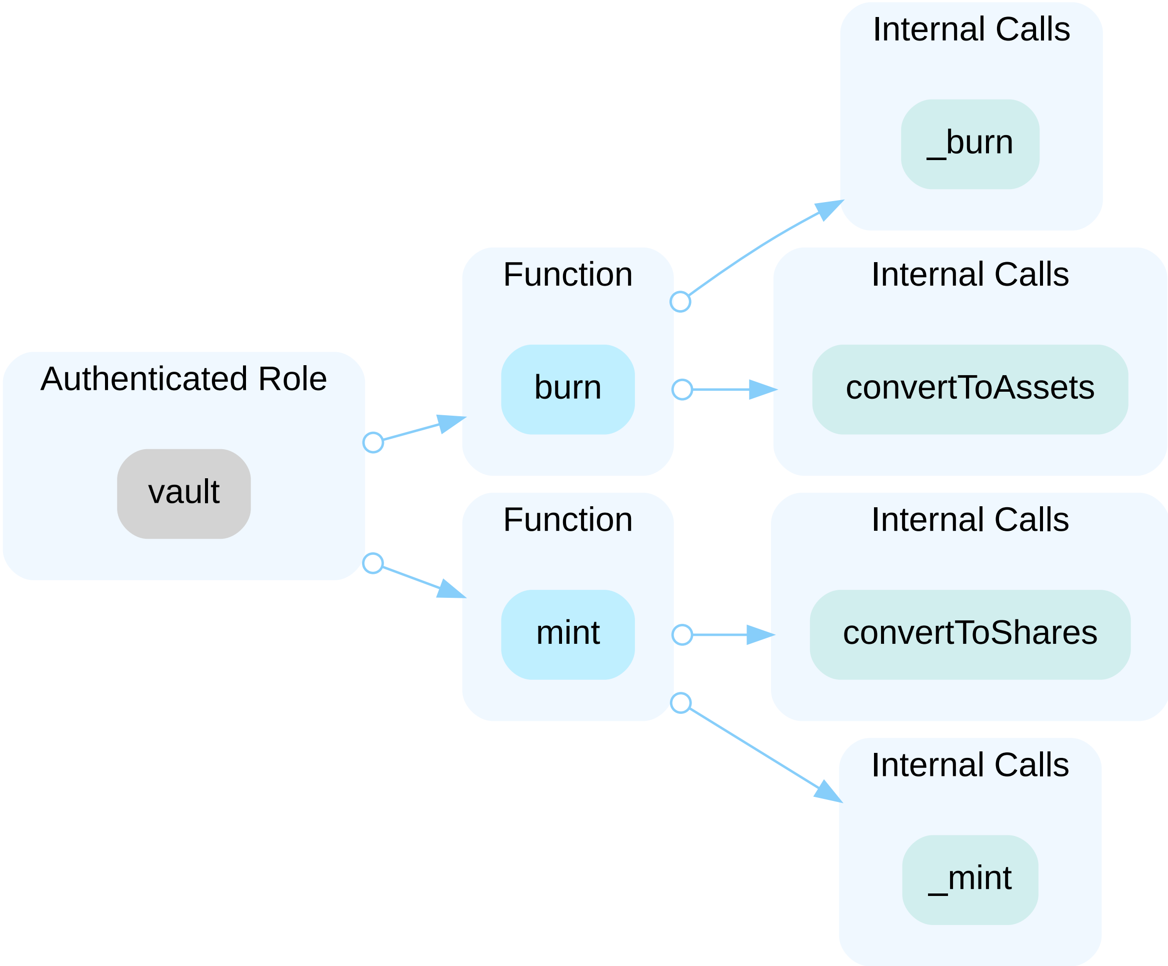


- In the contract `LpToken`, the role `_manager` of the `dataStorage` contract has authority over the function shown in the diagram below:

- In the contract `LpToken`, the role `_owner` of the `dataStorage` contract has authority over the functions shown in the diagram below:



- In the contract `LpToken`, the role `vault` has authority over the functions shown in the diagram below:



- In the contract `OwnableUpgradeable`, the role `_owner` has authority over the functions `transferOwnership()` and `renounceOwnership()`.

Design Considerations:

All Contracts Are Not Upgradeable

Although these contracts inherit upgradeable contracts, they are used directly rather than through proxies.

Minimum Deposit Requirement Can Be 0

The `deposit()` function in the `LRTVault` and `USDVault` contracts ensures that the deposit amount is not less than `dataStorage.minDepositMap(address(this))`:

```
  require(amount >= dataStorage.minDepositMap(address(this)), "Deposit amount too
small");
```

However, the minimum deposit value can be set to 0.

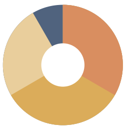
Vault Does Not Require Registration in DataStorage

While functions exist for the `_owner` to add and delete vaults from the `DataStorage` contract, vaults can still be used even if they are not added or have been deleted.

Vault Supports Redemption on Behalf of Others

The `redeemAndUnLockDeposit()` function is intended to unlock and redeem deposits for `msg.sender`. However, since the function does not verify that the provided "ids" belong to `msg.sender`, users can unlock deposits belonging to others, allowing those deposits to be redeemed without further action by their owners.

FINDINGS | GOLDEN GOOSE



12	0	4	4	3	1
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for Golden Goose. Through this audit, we have uncovered 12 issues ranging from different severity levels. Utilizing the techniques of Formal Verification, Manual Review & Static Analysis to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
GLOBAL-01	**Centralization Related Risks**	**Centralization**	**Major**	**Acknowledged**
LTR-01	Compiler Error In `LpToken` Contract	Coding Issue	Major	Resolved
USD-03	**Unprotected Principal In `USDVault` Contract**	**Centralization, Volatile Code**	**Major**	**Acknowledged**
USV-01	Incorrect Order Of `share` And `assetAmount` In `RedeemLock` Struct	Logical Issue	Major	Resolved
DSR-01	Minimum Deposit Value Not Cleared For Deleted Vault	Logical Issue	Medium	Resolved
LRT-01	Incorrect Token Balance Check In `moveToken()` Function Leads To Potential Transfer Failure	Logical Issue	Medium	Resolved
LTR-02	Incorrect Uint Type Used	Inconsistency	Medium	Resolved
USD-02	Potential Exploit In `getAvailableAmount()` Function Due To Lack Of ID Ownership Verification	Logical Issue	Medium	Resolved
LRV-01	Inaccurate Token Amount Recording In `getAvailableAmount` Function	Logical Issue	Minor	Acknowledged

ID	Title	Category	Severity	Status
RNA-03	Missing Zero Address Validation	Volatile Code	Minor	Resolved
RNA-04	Third-Party Dependency Usage	Design Issue	Minor	Acknowledged
CON-05	Unused Return Value	Volatile Code	Informational	Resolved

GLOBAL-01 | CENTRALIZATION RELATED RISKS

Category	Severity	Location	Status
Centralization	Major		Acknowledged

Description

In the contract `DataStorage` , the role `_owner` has authority over the following functions:

- `updateManager()`
- `updateDepositLockTime()`
- `updateRedeemLockTime()`
- `setVaultMinDeposit()`
- `addVault()`
- `delVault()`

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and update the manager address, update deposit lock time, delete a vault address, add a vault, set minimum deposit for vault, update the redeem lock time.

In the contract `LRTVault` , the role `_owner` of the `dataStorage` contract has authority over the following functions:

- `updateDataStorageContract()`
- `updateMaxLockAmount()`
- `moveToken()`
- `updateLockTime()`

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and update the data storage contract address, update the maximum lock amount, move tokens to a specified receiver address, and update the lock time (up to 180 days). It is important to note that the `_owner` can only transfer tokens that are not the deposited token type or deposited tokens exceeding the total staked amount.

In the contract `USDVault` , the role `_owner` of the `dataStorage` has authority over the following functions:

- `updatePreCustodian()`
- `updateDataStorageContract()`
- `updateCustodian()`
- `initialCustodian()`

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and update the `preCustodian` address, update the data storage contract address, update the custodian address, and set the initial custodian address.

In the contract `OwnableUpgradeable`, the role `_owner` has authority over the following functions:

- `transferOwnership()`
- `renounceOwnership()`

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and transfer or renounce the ownership.

In the contract `LpToken`, the role `_manager` of the `dataStorage` contract has authority over the following function:

- `updatePrice()`

Any compromise to the `_manager` account may allow the hacker to take advantage of this authority and update the price.

In the contract `LpToken`, the role `_owner` of the `dataStorage` contract has authority over the following functions:

- `forceUpdatePrice()`
- `updateDeviation()`

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and force update the price, update the maximum deviation setting.

In the contract `LpToken`, the role `vault` has authority over the following functions:

- `burn()`
- `mint()`

Any compromise to the `vault` account may allow the hacker to take advantage of this authority and mint or burn shares. It is important to note that for the entire project, LP Tokens are created by the `vault`, which is the `USDVault` contract. The standalone use of the `LpToken` contract is not intended, as all `LpToken` instances are created by the vault and are open-source.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Alleviation

[Golden Goose Team, 10/24/2024]:

The team acknowledged the issue and adopted the multisign solution to ensure the private key management process at the current stage. The `DataStorage` contract has transferred the ownership to a Gnosis Safe contract with 2/2 signers in the sensitive function signing process.

- `DataStorage` address: 0x857aB0b4F236F7DD7E5AC5F96C0bbEbF230c2D3B
- `_owner` address: 0x509B38c5F884067E2128c4FC89d1489813d695E0
- The multisign addresses:
    1. EOA:0x9bCAd39B7D70e7A57BEed4e1640AEEFe49bCa662

2. EOA:0xbE22D669EEb4B80Bd0568A833a82E29d007b9494

Additionally, the team implemented the following constraints:

1. In the `USDVault` contract, any change to the `custodian` requires first modifying the `preCustodian` and waiting 2 days before the change can take effect.
2. In the `LRTVault` contract, the `_owner` can set the maximum lock time, which is restricted to a maximum of 180 days.
3. In the `LpToken` contract, the `_owner` can set the price of the `LpToken` , but any price adjustment is capped at a 10% deviation and can only occur once every 12 hours.

[CertiK, 10/24/2024]:

While this strategy has indeed reduced the risk, it's crucial to note that it has not completely eliminated it. CertiK strongly encourages the project team to periodically revisit the private key security management of all above-listed addresses.

LTR-01 | COMPILER ERROR IN `LpToken` CONTRACT

Category	Severity	Location	Status
Coding Issue	Major	LpToken.sol (commit:be4456): 24	Resolved

Description

The `LPToken` contract fails to compile due to an incorrect number of arguments being passed to the `ERC20` constructor in its own constructor. The error message indicates that the `ERC20` constructor expects only two arguments (name and symbol), but three arguments (name, symbol, and decimals) are being provided.

Recommendation

Remove `decimals` from `ERC20(name,symbol,decimals)`.

Alleviation

[Golden Goose Team, 10/11/2024]: The team heeded the advice and resolved the issue in the updated code.

USD-03 | UNPROTECTED PRINCIPAL IN `USDVault` CONTRACT

Category	Severity	Location	Status
Centralization, Volatile Code	**Major**	**USDVault.sol (commit:0749dc): 60**	**Acknowledged**

Description

When the `deposit()` function is called, tokens are transferred to the address returned by `getCustodian()`, which is not the current contract address. The `withdraw()` function is designed to withdraw tokens from the contract. However, the contract does not guarantee that its token balance is always sufficient to cover the total staked amount. This leads to a situation where some users might not be able to withdraw their full principal.

Recommendation

Ensure that the contract maintains a sufficient token balance to cover all deposited amounts.

Alleviation

[Golden Goose Team, 09/26/2024]: After the user deposit, the token will enter custodian. After the user initiates a redeem, we will transfer token to the contract.

USV-01 | INCORRECT ORDER OF `share` AND `assetAmount` IN `RedeemLock` STRUCT

Category	Severity	Location	Status
Logical Issue	Major	USDVault.sol (commit:be4456): 147	Resolved

Description

In the `_redeem()` function, the `RedeemLock` struct is intended to store information about a redemption, including the `share` (LP token amount) and `assetAmount` (corresponding asset amount). However, the values for `share` and `assetAmount` are incorrectly reversed when generating the `RedeemLock`. This mistake could lead to various issues, such as failed redemptions or users receiving fewer tokens than expected.

Proof of Concept

Foundry test:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";

contract USDVaultSimplified {
    mapping(address => Deposit) private deposits;
    mapping(uint256 => RedeemLock) private redeemMap;
    mapping(address => uint256) public balances;

    struct Deposit{
        address account;
        uint256 assetAmount;
    }

    struct RedeemLock{
        address account;
        uint256 share;     // lp amount
        uint256 assetAmount;
    }

    constructor() {
        balances[msg.sender] = 100;
    }

    function depositAndUnLockDeposit(uint256 amount) public {
        balances[msg.sender] -= amount;
        deposits[msg.sender] = Deposit(msg.sender, 100);
    }

    function redeem() public {
        uint256 assetAmount = deposits[msg.sender].assetAmount;
        uint256 share = convertToShares(assetAmount);
        redeemMap[0] = RedeemLock(msg.sender,assetAmount,share);
    }

    function convertToShares(uint256 amount) public returns (uint256) {
        return amount / 2;
    }

    function withdraw() public {
        balances[msg.sender] += redeemMap[0].assetAmount;
    }
}

contract USDVaultSimplifiedTest is Test {
    USDVaultSimplified public vault;
    address public user = vm.addr(1);
```

```
    function setUp() public {
        vm.prank(user);
        vault = new USDVaultSimplified();
    }

    function testIssue() public {
        vm.startPrank(user);
        require(vault.balances(user) == 100, "error1"); // the user's initial
balance is 100
        vault.depositAndUnLockDeposit(100);
        vault.redeem();
        vault.withdraw();
        require(vault.balances(user) == 50, "error2"); // the user's current balance
is 50 (shares), rather than 100 (assetAmount)
        vm.stopPrank();
    }
}
```

Recommendation

Swap the `share` and `assetAmount` values when creating the `RedeemLock` struct in the `_redeem()` function to correctly assign them to their respective fields.

Alleviation

[Golden Goose Team, 10/11/2024]: The team heeded the advice and resolved the issue in the updated code.

DSR-01 | MINIMUM DEPOSIT VALUE NOT CLEARED FOR DELETED VAULT

Category	Severity	Location	Status
Logical Issue	● Medium	DataStorage.sol (commit:0749dc): 46	● Resolved

Description

The function `delVault()` is called by the manager to delete the specified vault. The variable `minDepositMap` is intended to store the minimum deposit value for this vault. If the vault is deleted, this minimum deposit value should be reset to 0, indicating that there are no deposit limits. However, `minDepositMap` is not set to 0.

Recommendation

In the `delVault` function, ensure that after a vault is deleted, the corresponding `minDepositMap` value is set to 0 to eliminate unnecessary deposit restrictions.

Alleviation

[Golden Goose Team, 09/26/2024]: The team heeded the advice and resolved the issue in the updated code.

LRT-01 | INCORRECT TOKEN BALANCE CHECK IN `moveToken()` FUNCTION LEADS TO POTENTIAL TRANSFER FAILURE

Category	Severity	Location	Status
Logical Issue	Medium	LRTVault.sol (commit:0749dc): 57	Resolved

Description

The `moveToken()` function in the `LRTVault` contract is designed to transfer excess tokens from the contract. When `tokenContract` is not the same as `token`, the function should read the balance of the `tokenContract` and transfer the specified amount of tokens. However, the function mistakenly reads the balance of `token` instead of `tokenContract`, which may result in failed or incomplete token transfers when trying to move tokens.

Recommendation

Modify the function to correctly read the balance of `tokenContract` when `tokenContract` is not equal to `token`.

Alleviation

[Golden Goose Team, 09/26/2024]: The team heeded the advice and resolved the issue in the updated code.

LTR-02 | INCORRECT UINT TYPE USED

Category	Severity	Location	Status
Inconsistency	Medium	LpToken.sol (commit:be4456): 17, 24	Resolved

Description

Upon initialization, the `PRIRCE_DECIMAL` variable is of `uint256` . However, in the `constructor` function, `decimals` is assigned a `uint8` , which may be too small.

Recommendation

We recommend using the same uint type to update the variable.

Alleviation

[Golden Goose Team, 10/11/2024]: The team heeded the advice and resolved the issue in the updated code.

USD-02 | POTENTIAL EXPLOIT IN `getAvailableAmount()` FUNCTION DUE TO LACK OF ID OWNERSHIP VERIFICATION

Category	Severity	Location	Status
Logical Issue	Medium	USDVault.sol (commit:0749dc): 158	Resolved

Description

The `getAvailableAmount()` function is intended to calculate the total redeemed and deposited token amount for a given `account` based on the `ids` in the `depositMap`. However, the function does not verify whether the provided `ids` actually belong to the `account`. This oversight could allow a hacker to pass arbitrary `ids` and receive a larger amount than expected.

Recommendation

Modify the function to check if each `id` in the `depositMap` actually belongs to the `account` before including it in the calculation.

Alleviation

[Golden Goose Team, 09/26/2024]: The team heeded the advice and resolved the issue in the updated code.

LRV-01 | INACCURATE TOKEN AMOUNT RECORDING IN `getAvailableAmount` FUNCTION

Category	Severity	Location	Status
Logical Issue	Minor	LRTVault.sol (commit:a68c57): 158	Acknowledged

Description

The `getAvailableAmount()` function is designed to return the total amount of tokens a user has deposited into the system. However, the implementation contains a logical flaw in line 158: rather than accumulating the token balances across multiple deposits, the function overwrites the balance with the value associated with the most recent `ids` passed as a parameter. As a result, users querying their balance will receive an incomplete or incorrect amount, reflecting only their latest deposit instead of the full balance accumulated across multiple `ids`. This issue can lead to inaccurate balance queries.

Recommendation

Refactor the function to accumulate the token amounts across all deposits associated with the user's `ids`. The logic should ensure that the function sums the total deposit amount and returns it accurately.

Alleviation

[Golden Goose Team, 10/17/2024]: The team acknowledged the finding and decided not to change the current codebase.

RNA-03 | MISSING ZERO ADDRESS VALIDATION

Category	Severity	Location	Status
Volatile Code	Minor	DataStorage.sol (commit:0749dc): 21, 51; LRTVault.sol (commit:0749dc): 25; USDVault.sol (commit:0749dc): 48, 170; VaultFactory.sol (commit:0749dc): 16	Resolved

Description

Addresses are not validated before assignment or external calls, potentially allowing the use of zero addresses and leading to unexpected behavior or vulnerabilities. For example, transferring tokens to a zero address can result in a permanent loss of those tokens.

- `initialManager` is not zero-checked before being used.
- `account` is not zero-checked before being used.
- `storageContract` is not zero-checked before being used.
- `tokenContract` is not zero-checked before being used.
- `account` is not zero-checked before being used.
- `tokenContract` is not zero-checked before being used.

Recommendation

It is recommended to add a zero-check for the passed-in address value to prevent unexpected errors.

Alleviation

[Golden Goose Team, 09/26/2024]: The team heeded the advice and resolved the issue in the updated code.

RNA-04 | THIRD-PARTY DEPENDENCY USAGE

Category	Severity	Location	Status
Design Issue	Minor	LRTVault.sol (commit:0749dc): 14, 57; USDVault.sol (commit:0749dc): 15, 16; interfaces/IDataStorage.sol (commit:0749dc): 4~19	Acknowledged

Description

The contract is serving as the underlying entity to interact with one or more third party protocols. The scope of the audit treats third party entities as black boxes and assumes their functional correctness. However, in the real world, third parties can be compromised and this may lead to lost or stolen assets. In addition, upgrades of third parties can possibly create severe impacts, such as increasing fees of third parties, migrating to new LP pools, etc.

- The function `LRTVault.moveToken` interacts with third party contract with `IERC20` interface via `tokenContract`.
- The contract `LRTVault` interacts with third party contract with `IERC20` interface via `token`.
- The contract `USDVault` interacts with third party contract with `IERC20` interface via `token`.
- The contract `USDVault` interacts with `custodian`.

Recommendation

The auditors understood that the business logic requires interaction with third parties. item_output is recommended for the team to constantly monitor the statuses of third parties to mitigate the side effects when unexpected activities are observed.

Alleviation

[Golden Goose Team, 09/26/2024]: The team acknowledged the finding and decided not to change the current codebase.

CON-05 | UNUSED RETURN VALUE

Category	Severity	Location	Status
Volatile Code	Informational	LRTVault.sol (commit:be4456): 79; USDVault.sol (commit:be4456): 191~193	Resolved

Description

The smart contract does not check or store the return value of an external call in a local or state variable, which may introduce vulnerabilities due to the unhandled outcome. We would like to know the intended design.

Recommendation

We would like to confirm with the client if the current implementation aligns with the original project design.

Alleviation

[Golden Goose Team, 10/17/2024]: The team heeded the advice and resolved the issue in the updated code.

FORMAL VERIFICATION | GOLDEN GOOSE

Formal guarantees about the behavior of smart contracts can be obtained by reasoning about properties relating to the entire contract (e.g. contract invariants) or to specific functions of the contract. Once such properties are proven to be valid, they guarantee that the contract behaves as specified by the property. As part of this audit, we applied formal verification to prove that important functions in the smart contracts adhere to their expected behaviors.

Considered Functions And Scope

In the following, we provide a description of the properties that have been used in this audit. They are grouped according to the type of contract they apply to.

Verification of Standard Ownable Properties

We verified *partial* properties of the public interfaces of those token contracts that implement the Ownable interface. This involves:

- function `owner` that returns the current owner,
- functions `renounceOwnership` that removes ownership,
- function `transferOwnership` that transfers the ownership to a new owner.

The properties that were considered within the scope of this audit are as follows:

Property Name	Title
ownable-transferownership-correct	Ownership is Transferred
ownable-owner-succeed-normal	`owner` Always Succeeds
ownable-renounce-ownership-is-permanent	Once Renounced, Ownership Cannot be Regained
ownable-renounceownership-correct	Ownership is Removed

Verification Results

For the following contracts, formal verification established that each of the properties that were in scope of this audit (see scope) are valid:

Detailed Results For Contract DataStorage (projects/GoldenGoose/contracts/DataStorage.sol) In SHA256 Checksum f1ceb3243b905ef51431b49717cbb27360a94c69

Verification of Standard Ownable Properties

Detailed Results for Function `transferOwnership`

Property Name	Final Result	Remarks
ownable-transferownership-correct	● True	

Detailed Results for Function `owner`

Property Name	Final Result	Remarks
ownable-owner-succeed-normal	● True	

Detailed Results for Function `renounceOwnership`

Property Name	Final Result	Remarks
ownable-renounce-ownership-is-permanent	● True	
ownable-renounceownership-correct	● True	

APPENDIX | GOLDEN GOOSE

Finding Categories

Categories	Description
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

Details on Formal Verification

Some Solidity smart contracts from this project have been formally verified. Each such contract was compiled into a mathematical model that reflects all its possible behaviors with respect to the property. The model takes into account the semantics of the Solidity instructions found in the contract. All verification results that we report are based on that model.

The following assumptions and simplifications apply to our model:

- Certain low-level calls and inline assembly are not supported and may lead to a contract not being formally verified.
- We model the semantics of the Solidity source code and not the semantics of the EVM bytecode in a compiled contract.

Formalism for property specifications

All properties are expressed in a behavioral interface specification language that CertiK has developed for Solidity, which allows us to specify the behavior of each function in terms of the contract state and its parameters and return values, as well as contract properties that are maintained by every observable state transition. Observable state transitions occur when the contract's external interface is invoked and the invocation does not revert, and when the contract's Ether balance is changed by the EVM due to another contract's "self-destruct" invocation. The specification language has the usual Boolean connectives, as well as the operator `\old` (used to denote the state of a variable before a state transition), and several types of specification clause:

Apart from the Boolean connectives and the modal operators "always" (written `[]`) and "eventually" (written `<>`), we use the following predicates to reason about the validity of atomic propositions. They are evaluated on the contract's state whenever a discrete time step occurs:

- `requires [cond]` - the condition `cond`, which refers to a function's parameters, return values, and contract state variables, must hold when a function is invoked in order for it to exhibit a specified behavior.
- `ensures [cond]` - the condition `cond`, which refers to a function's parameters, return values, and both `\old` and current contract state variables, is guaranteed to hold when a function returns if the corresponding requires condition held when it was invoked.
- `invariant [cond]` - the condition `cond`, which refers only to contract state variables, is guaranteed to hold at every observable contract state.
- `constraint [cond]` - the condition `cond`, which refers to both `\old` and current contract state variables, is guaranteed to hold at every observable contract state except for the initial state after construction (because there is no previous state); constraints are used to restrict how contract state can change over time.

Description of the Analyzed Ownable Properties

Properties related to function `transferOwnership`

ownable-transferownership-correct

Invocations of `transferOwnership(newOwner)` must transfer the ownership to the `newOwner`.

Specification:

```
ensures this.owner() == newOwner;
```

Properties related to function `owner`

ownable-owner-succeed-normal

Function `owner` must always succeed if it does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `renounceOwnership`

ownable-renounce-ownership-is-permanent

The contract must prohibit regaining of ownership once it has been renounced.

Specification:

```
constraint \old(owner()) == address(0) ==> owner() == address(0);
```

ownable-renounceownership-correct

Invocations of `renounceOwnership()` must set ownership to address(0).

Specification:

```
ensures this.owner() == address(0);
```

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK’s prior written consent in each instance.

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED “AS IS” AND “AS AVAILABLE” AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER’S OR ANY OTHER PERSON’S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

CertiK | **Securing** the **Web3** World

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.