



Security Assessment

ORIGIN

CertiK Assessed on Apr 22nd, 2024





Certik Assessed on Apr 22nd, 2024

ORIGIN

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

Executive Summary

TYPES

DeFi

ECOSYSTEM

Binance Smart Chain
(BSC) | Ethereum (ETH)

METHODS

Formal Verification, Manual Review, Static Analysis

LANGUAGE

Solidity

TIMELINE

Delivered on 04/22/2024

KEY COMPONENTS

N/A

CODEBASE

Privately shared

[View All in Codebase Page](#)

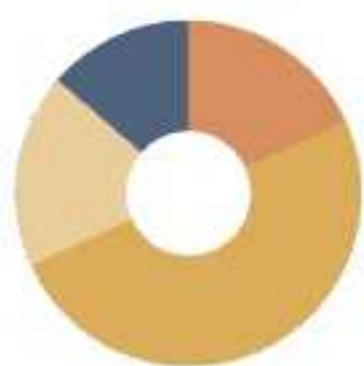
Highlighted Centralization Risks

⚠️ Privileged role can mint tokens

⚠️ Fees are bounded by 100%

⚠️ Has blacklist/whitelist

Vulnerability Summary



22

Total Findings

11

Resolved

0

Mitigated

0

Partially Resolved

11

Acknowledged

0

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.

11

Medium

5 Resolved, 6 Acknowledged

Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.

4

Minor

4 Resolved

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.

3

Informational

3 Acknowledged

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 | ORIGIN

Summary

[Executive Summary](#)

[Vulnerability Summary](#)

[Codebase](#)

[Audit Scope](#)

[Approach & Methods](#)

Findings

[GLOBAL-01 : Centralization Related Risks](#)

[ORG-01 : Ownership Can be Regained After Renouncement](#)

[ORN-02 : Loss of Staked Tokens for Users Due to the `BondDepositoryDai` and `StakingHelper` Implementation](#)

[TOR-02 : Manager role in Treasury could transfer out token asset in contract with a malicious `curve` address](#)

[BDO-01 : Risk of Bond Price Manipulation due to Uncontrolled `setAdjustment\(\)` Calls](#)

[BDO-05 : Miscalculation of Profit due to Incorrect `\\_inviteProfit` Handling](#)

[CKP-01 : Incompatibility With Deflationary Tokens](#)

[ERC-02 : No upper limits for `feeRatio` and `buyFeeRatio`](#)

[ETH-01 : Missing staleness checks for Chainlink oracle](#)

[ORG-04 : Anyone Can Call `claim\(\)`/`redeem\(\)` for Any Arbitrary `recipient`](#)

[ORG-08 : Out-of-Scope Dependencies](#)

[ORN-01 : Compilation Errors in `WETHBondDepository` and `BondDepositoryDai` Contracts](#)

[ORN-03 : Incorrect `StakingHelper` Contract Version Used in `BondDepositoryDai` and `WETHBondDepository`](#)

[SDO-01 : Incorrect Minting Amounts Due to Dynamic `totalSupply`](#)

[TOR-01 : Potential Exploitation via `sOHM` Transfers and Debt Mechanism in `Treasury` Contract](#)

[BDO-02 : `totalDebt` Could Exceed `maxDebt`](#)

[ETH-02 : Incorrect event parameters](#)

[ORG-05 : Susceptible to Signature Malleability](#)

[ORG-06 : Possibility of Cross Chain Replay Attack in `permit\(\)` Function](#)

[BDO-04 : Deprecated Usage of `Counters.sol`](#)

[CKP-02 : Function Misses Zero Address Validation](#)

[ERC-01 : Function `\\_burnFrom\(\)` is Public](#)

Formal Verification

Considered Functions And Scope

Verification Results

■ **Appendix**

■ **Disclaimer**

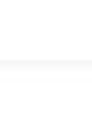
CODEBASE | ORIGIN

Repository

Privately shared

AUDIT SCOPE | ORIGIN

26 files audited ● 5 files with Acknowledged findings ● 1 file with Resolved findings ● 20 files without findings

ID	Repo	File	SHA256 Checksum
● BDR	CertiKProject/certik-audit-projects	 1/BondDepository.sol	943e1a3a1d93c246cc6bb1d4a4ddad4cf3a d3fce29db9932a212300075b32fb1
● SOI	CertiKProject/certik-audit-projects	 1/Staking.sol	98d40341a6de9e8e3f8fc8c22ebc1a2fb7fa d191dd406ac73b09044b973fbb2d
● SHR	CertiKProject/certik-audit-projects	 1/StakingHelper.sol	2443064312fd8d593a301db2aff56ade602a c972c712c235d511295466aa9cea
● ERI	CertiKProject/certik-audit-projects	 1/sERC20.sol	4bc7aa9faa234849a865f40717b9c0c5cf29 50bc61d94a92f998b6d7e2812eaa
● ERO	CertiKProject/certik-audit-projects	 1/sERC20.sol	28d5311b890c68755d0c889e7ccb2683b9f 1ff5f4905f04d37a3d7bd27429b03
● ETB	CertiKProject/certik-audit-projects	 1/wETHBondDepository.sol	86be1b65fcee3ed2c78ebb9107c3c281663f 7efcd4ad83ef91c2b1d916cb136f
● ERR	CertiKProject/certik-audit-projects	 1/ERC20.sol	088c2c9cff57162d16a4b88da3bd1b8336ff6 c748ba88962e2398df6628c3fb2
● RHR	CertiKProject/certik-audit-projects	 1/RedeemHelper.sol	a7d31d2be8151bf037d2cf8bf0b097e2ce2c eab2cad40de77ed70bd4f9879304
● SDR	CertiKProject/certik-audit-projects	 1/StakingDistributor.sol	48e078371a3a43ac6fbcf372dfb94a007a51 977af7d7390cc7e7ffb79256249b
● SWR	CertiKProject/certik-audit-projects	 1/StakingWarmup.sol	db8a28b7127a7358c975d8b14590f891b02 f5d37392582846b4450bb8b6e3b3c
● SBO	CertiKProject/certik-audit-projects	 1/StandardBondingCalculator.sol	3064a9bda43d33c31d5de66be41a7328f55 f0525bac38d119dda12743e6af51f
● TOI	CertiKProject/certik-audit-projects	 2/Treasury.sol	80f10e99b17c341218ae8a27f841b471e87 cc7b18f9a44cf37588a5153407937
● BDO	CertiKProject/certik-audit-projects	 1/BondDepository.sol	4974b83dce74de75758cb709b0bfebaf74bf 456cd2dac17596b365fae415ca92

ID	Repo	File	SHA256 Checksum
● ERC	CertiKProject/certik-audit-projects	 1/ERC20.sol	7801a29eb85adb849b55e56fb636d215285f196c6313a2515fcf271b25c26caa
● RHO	CertiKProject/certik-audit-projects	 1/RedeemHelper.sol	043e1e7800cc766c857c6075f27aa6ae5d5ea846570c64f03553993d4e164dc4
● SOR	CertiKProject/certik-audit-projects	 1/Staking.sol	281791aaa27b67eeacfb5b3678fd76ff18b5d78358030be8e879eb4d2773ca4b
● SDO	CertiKProject/certik-audit-projects	 1/StakingDistributor.sol	03d2c4cb5e9f68aac12b7fab325d36c50c62c9b985057cfb4d39349372837910
● SHO	CertiKProject/certik-audit-projects	 1/StakingHelper.sol	2443064312fd8d593a301db2aff56ade602ac972c712c235d511295466aa9cea
● SWO	CertiKProject/certik-audit-projects	 1/StakingWarmup.sol	db8a28b7127a7358c975d8b14590f891b02f5d37392582846b4450bb8b6e3b3c
● SBC	CertiKProject/certik-audit-projects	 1/StandardBondingCalculator.sol	3064a9bda43d33c31d5de66be41a7328f55f0525bac38d119dda12743e6af51f
● ETH	CertiKProject/certik-audit-projects	 1/wETHBondDepository.sol	4535c5458ebb1f09a74a3a6a37f19d430f3b62a2ba5214253552e58a8d701ae3
● TOR	CertiKProject/certik-audit-projects	 2/Treasury.sol	80f10e99b17c341218ae8a27f841b471e87cc7b18f9a44cf37588a5153407937
● ETD	CertiKProject/certik-audit-projects	 1/wETHBondDepository.sol	bb40400212515c964685eea7592d2828f4504701bb6f885cf7b78b79a61dac87
● BDI	CertiKProject/certik-audit-projects	 1/BondDepository.sol	b6fd1858dad9d7f524bc3a691457370553196fe0f20fba4952d17dc25b92a96d
● SHI	CertiKProject/certik-audit-projects	 1/StakingHelper.sol	61be0dc253a5891801b5b00133b9752fb1bfe62453fb07c847be9761c844e023
● ETO	CertiKProject/certik-audit-projects	 1/wETHBondDepository.sol	e6b9c63b5cfc4a9e40de10782e301fe6f91013e19b987f0162c5102144e0d15b

APPROACH & METHODS | ORIGIN

This report has been prepared for ORIGIN to discover issues and vulnerabilities in the source code of the ORIGIN project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing 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.

FINDINGS | ORIGIN



22

Total Findings

0

Critical

4

Major

11

Medium

4

Minor

3

Informational

This report has been prepared to discover issues and vulnerabilities for ORIGIN. Through this audit, we have uncovered 22 issues ranging from different severity levels. Utilizing the techniques of 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
ORG-01	Ownership Can Be Regained After Renouncement	Logical Issue	Major	● Resolved
ORN-02	Loss Of Staked Tokens For Users Due To The <code>BondDepositoryDai</code> And <code>StakingHelper</code> Implementation	Logical Issue	Major	● Resolved
TOR-02	Manager Role In Treasury Could Transfer Out Token Asset In Contract With A Malicious <code>curve</code> Address	Centralization	Major	● Acknowledged
BDO-01	Risk Of Bond Price Manipulation Due To Uncontrolled <code>setAdjustment()</code> Calls	Logical Issue	Medium	● Resolved
BDO-05	Miscalculation Of Profit Due To Incorrect <code>_inviteProfit</code> Handling	Logical Issue	Medium	● Resolved
CKP-01	Incompatibility With Deflationary Tokens	Logical Issue	Medium	● Acknowledged
ERC-02	No Upper Limits For <code>feeRatio</code> And <code>buyFeeRatio</code>	Logical Issue	Medium	● Acknowledged
ETH-01	Missing Staleness Checks For Chainlink Oracle	Logical Issue	Medium	● Resolved
ORG-04	Anyone Can Call <code>claim()</code> / <code>redeem()</code> For Any Arbitrary <code>_recipient</code>	Logical Issue	Medium	● Acknowledged

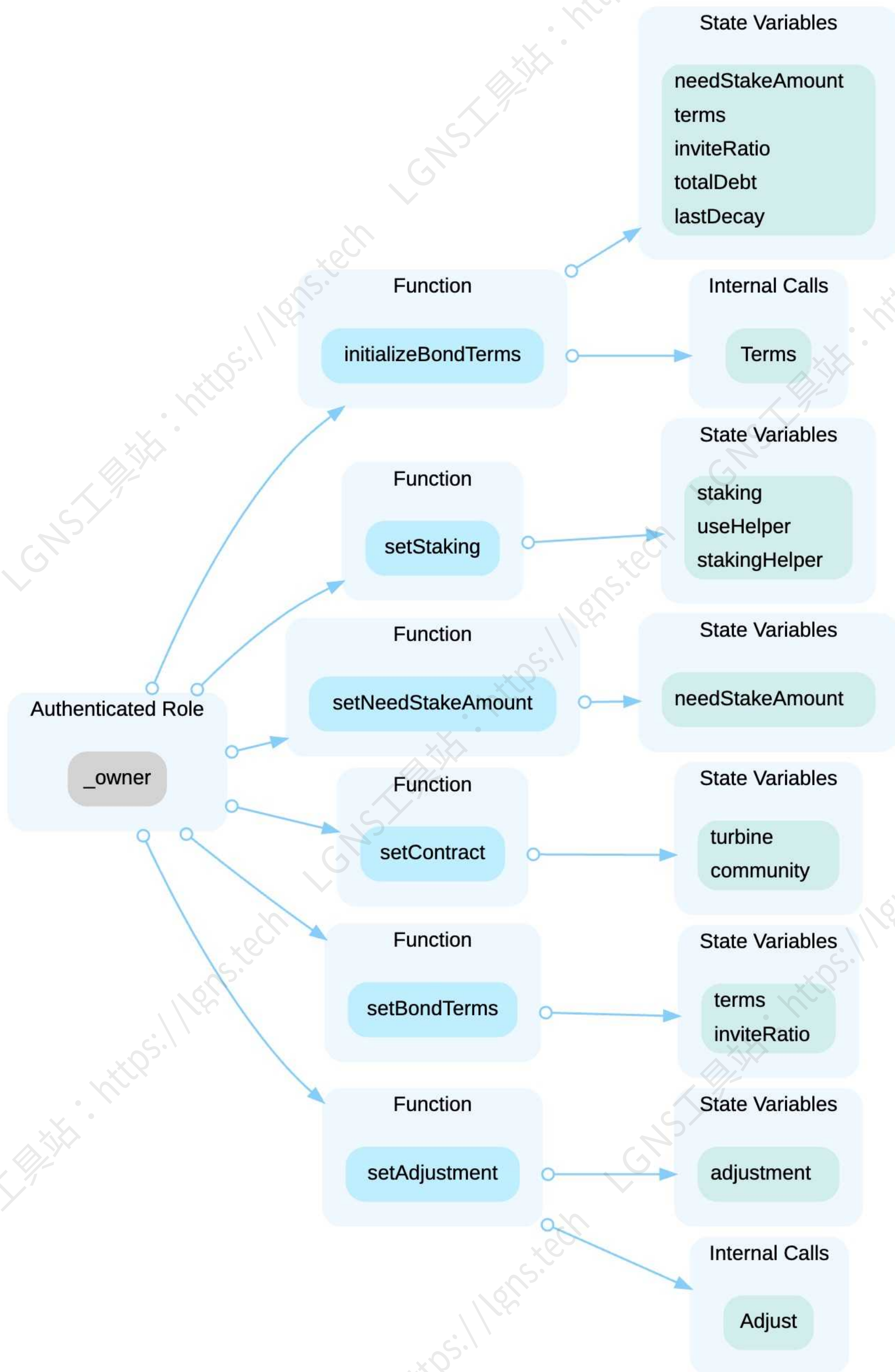
ID	Title	Category	Severity	Status
ORG-08	Out-Of-Scope Dependencies	Volatile Code	Medium	● Acknowledged
ORN-01	Compilation Errors In <code>WETHBondDepository</code> And <code>BondDepositoryDai</code> Contracts	Coding Issue	Medium	● Resolved
ORN-03	Incorrect <code>StakingHelper</code> Contract Version Used In <code>BondDepositoryDai</code> And <code>WETHBondDepository</code>	Design Issue	Medium	● Resolved
SDO-01	Incorrect Minting Amounts Due To Dynamic <code>totalSupply</code>	Logical Issue	Medium	● Acknowledged
TOR-01	Potential Exploitation Via <code>sOHM</code> Transfers And Debt Mechanism In <code>Treasury</code> Contract	Design Issue	Medium	● Acknowledged
BDO-02	<code>totalDebt</code> Could Exceed <code>maxDebt</code>	Logical Issue	Minor	● Resolved
ETH-02	Incorrect Event Parameters	Logical Issue	Minor	● Resolved
ORG-05	Susceptible To Signature Malleability	Volatile Code	Minor	● Resolved
ORG-06	Possibility Of Cross Chain Replay Attack In <code>permit()</code> Function	Volatile Code	Minor	● Resolved
BDO-04	Deprecated Usage Of <code>Counters.sol</code>	Logical Issue	Informational	● Acknowledged
CKP-02	Function Misses Zero Address Validation	Logical Issue	Informational	● Acknowledged
ERC-01	Function <code>_burnFrom()</code> Is Public	Design Issue	Informational	● Acknowledged

GLOBAL-01 | CENTRALIZATION RELATED RISKS

Category	Severity	Location	Status
Centralization	● Major		● Acknowledged

Description

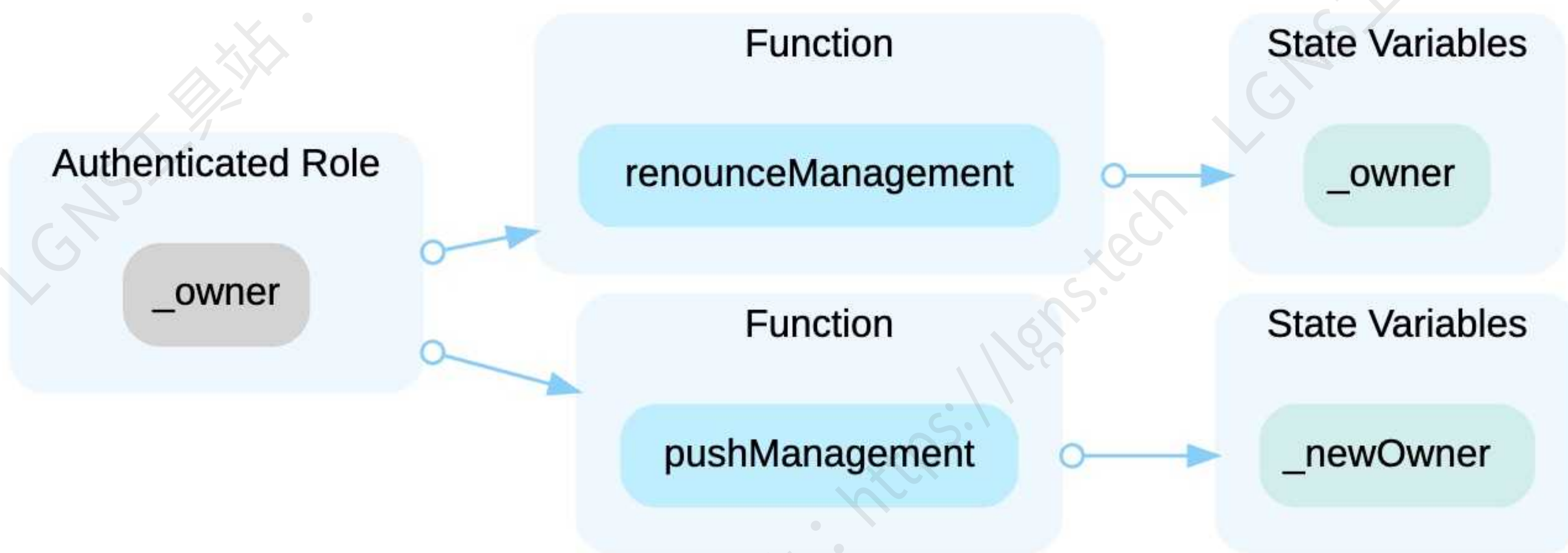
In the contract `BondDepositoryDai` the role `_owner` has authority over the functions shown in the diagram below. Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and modify bond settings and term arguments.



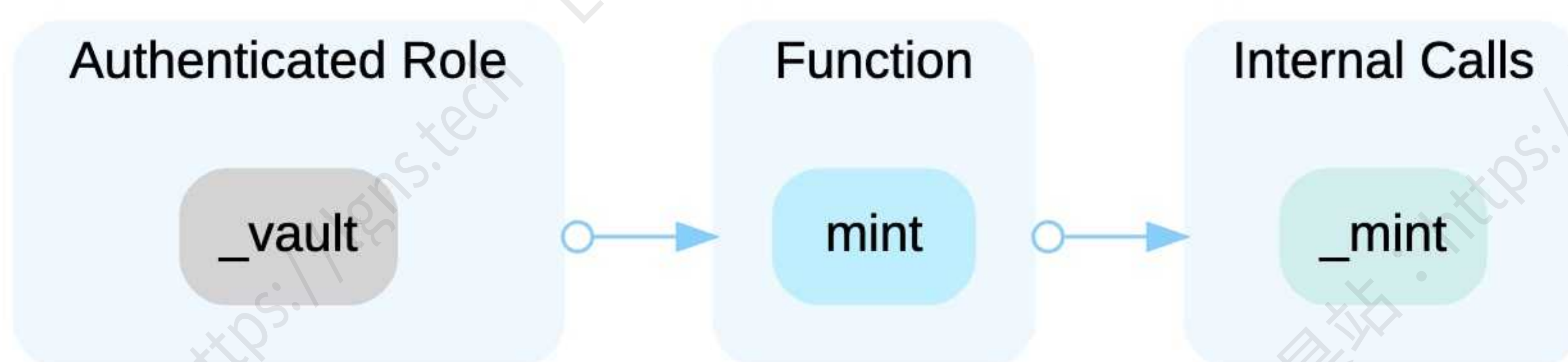
In the contract `Ownable` the role `_newOwner` has authority over the functions shown in the diagram below. Any compromise to the `_newOwner` account may allow the hacker to take advantage of this authority and retrieve contract ownership, **even if the ownership has been renounced**.



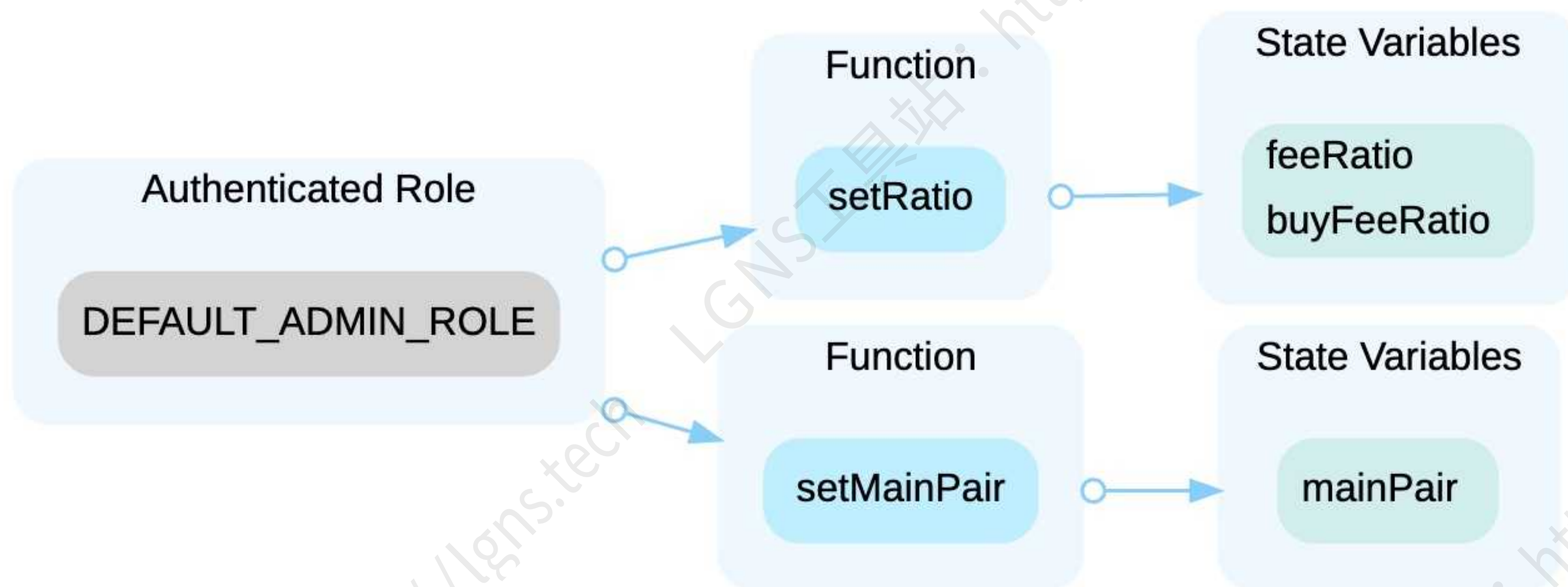
In the contract `Ownable` the role `_owner` has authority over the functions shown in the diagram below. Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and renounce/transfer contract ownership to arbitrary accounts.



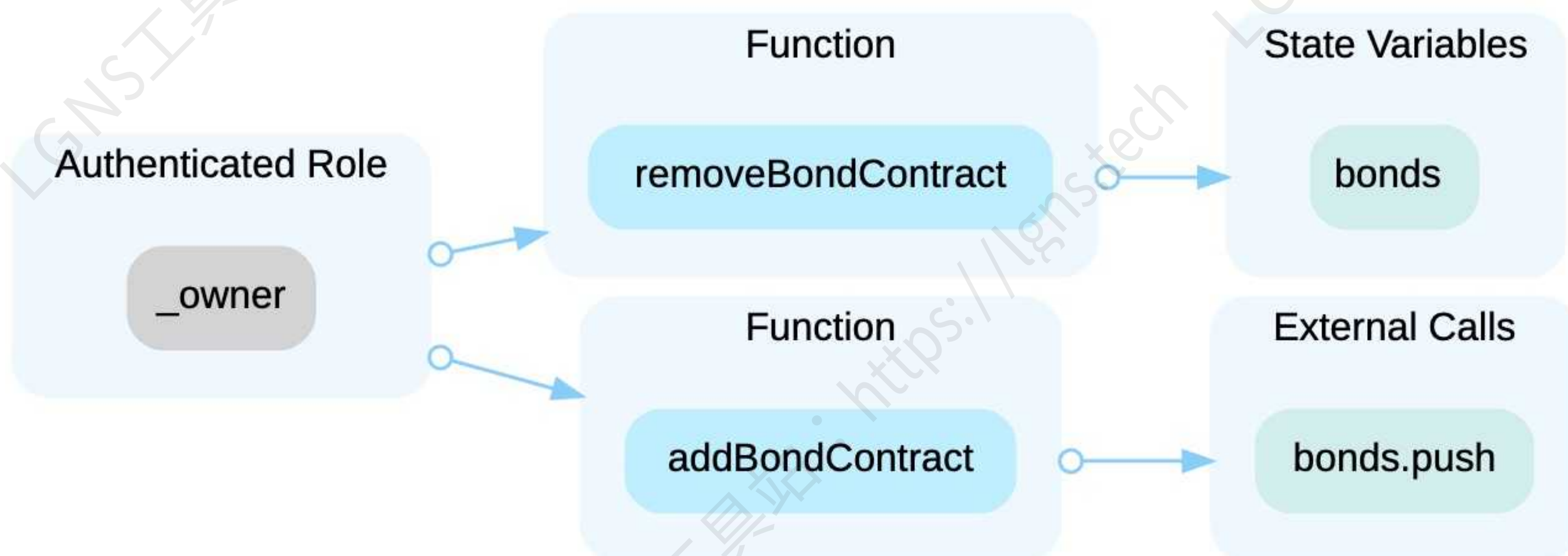
In the contract `OriginERC20Token` the role `MINT` has authority over the functions shown in the diagram below. Any compromise to the `MINT` role may allow the hacker to take advantage of this authority and **mint arbitrary ERC-20 tokens to any account**.



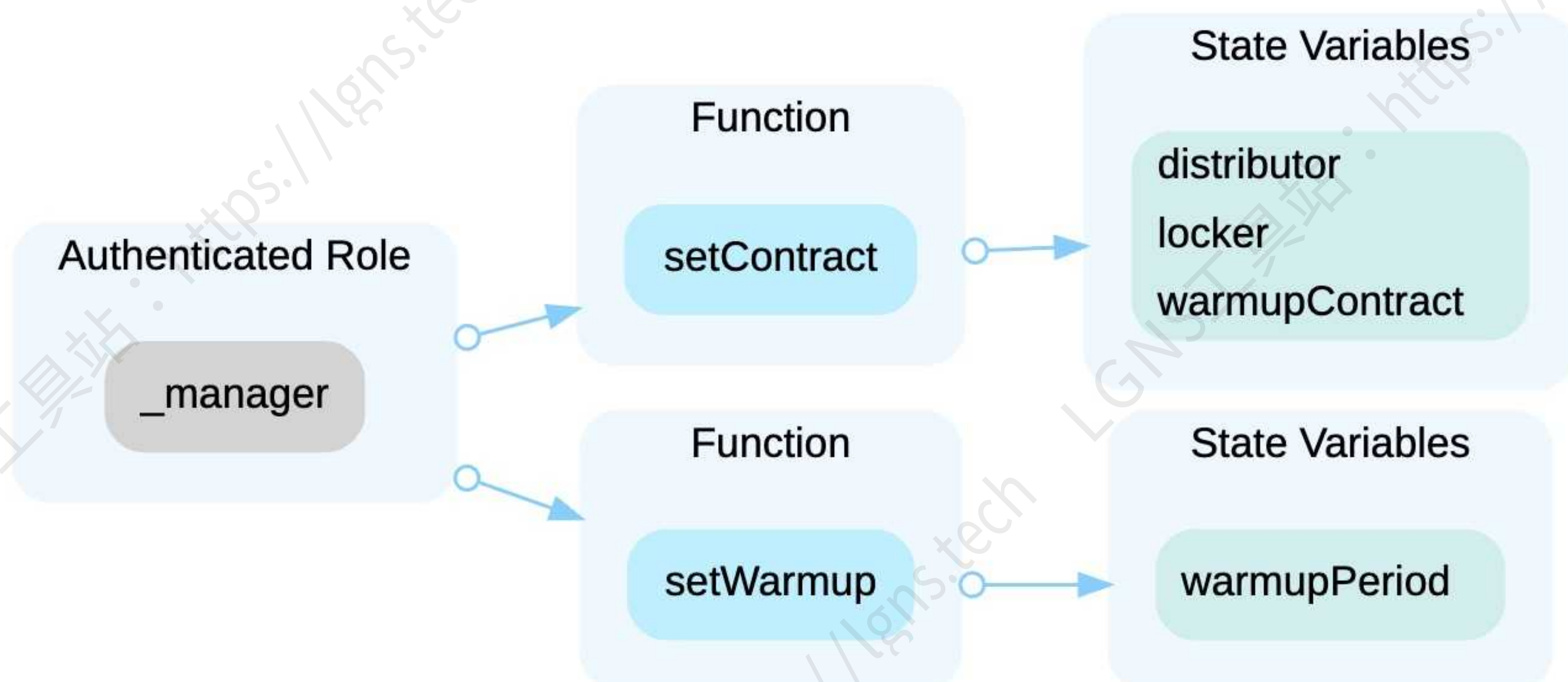
In the contract `OriginWithLottery` the role `DEFAULT_ADMIN_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow the hacker to take advantage of this authority and set the main Dex pair address and set the buy/sell fee ratio.



In the contract `RedeemHelper` the role `_owner` has authority over the functions shown in the diagram below. Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and modify bond contract addresses that are supported by the `RedeemHelper`.

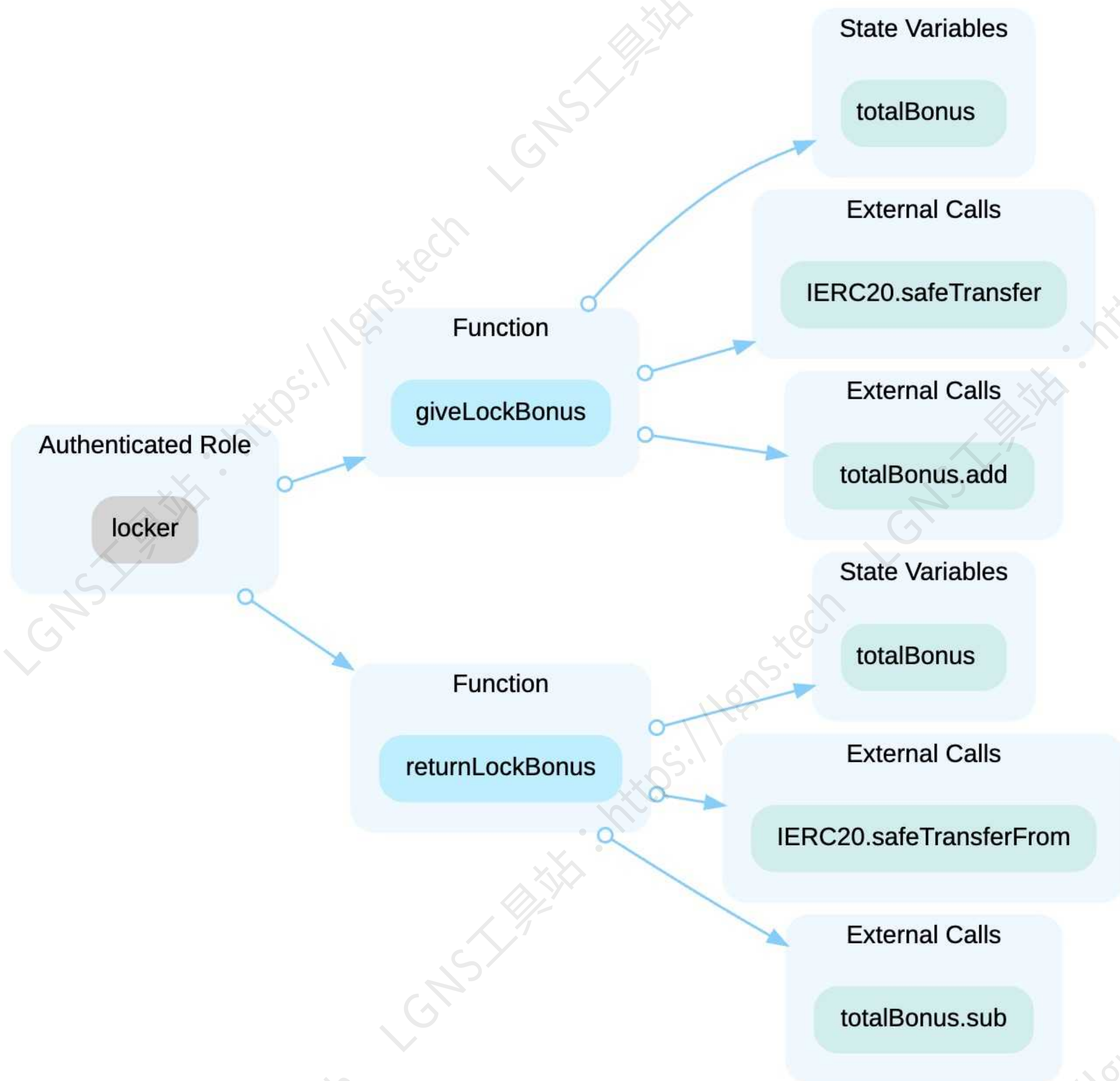


In the contract `Staking` the role `_manager` has authority over the functions shown in the diagram below. Any compromise to the `_manager` account may allow the hacker to take advantage of this authority and modify dependency contract addresses and the `warmupPeriod`.

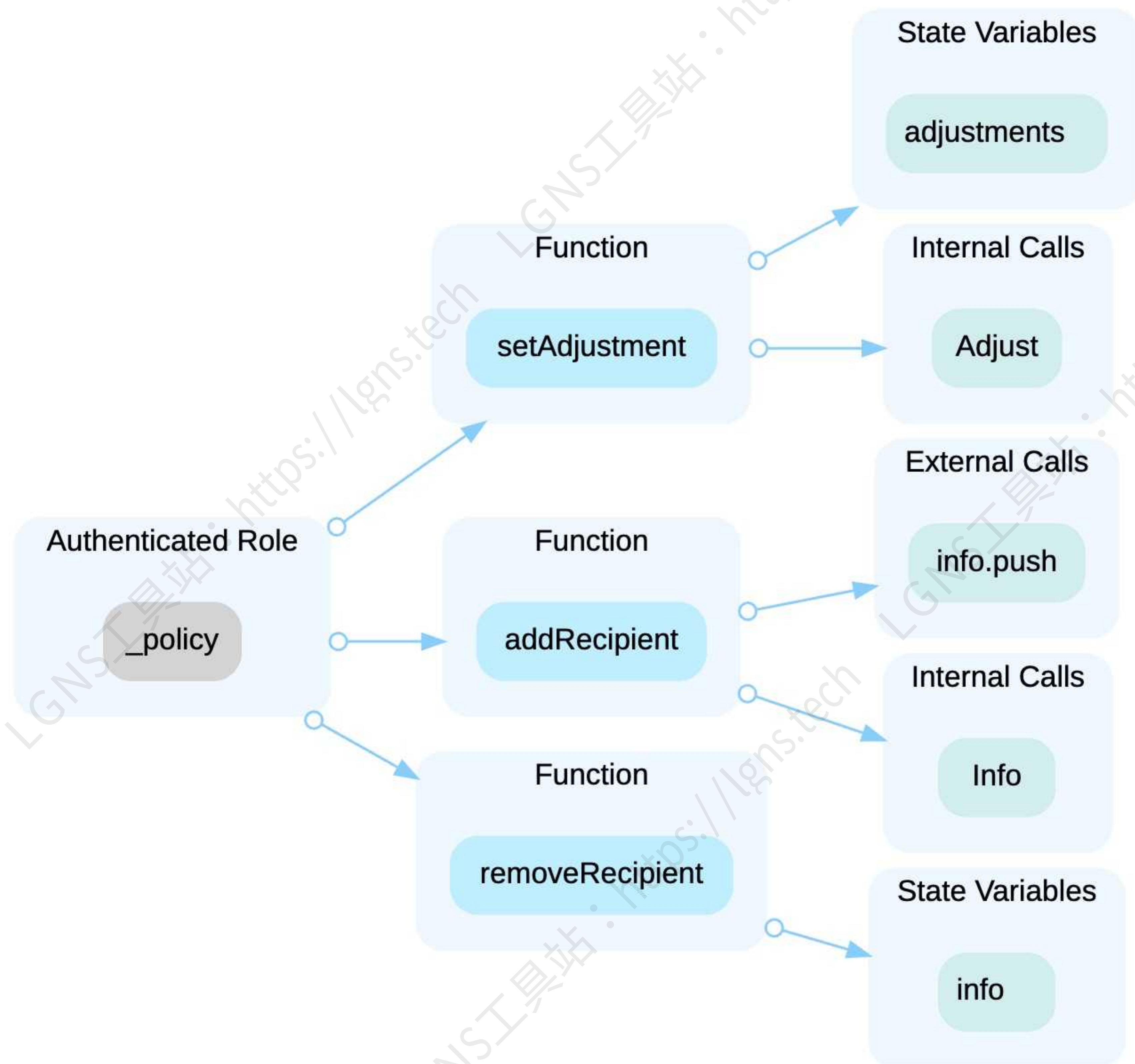


In the contract `Staking` the role `locker` has authority over the functions `giveLockBonus()` and `returnLockBonus`. Any compromise to the `locker` account may allow the hacker to take advantage of this authority and **transfer arbitrary**

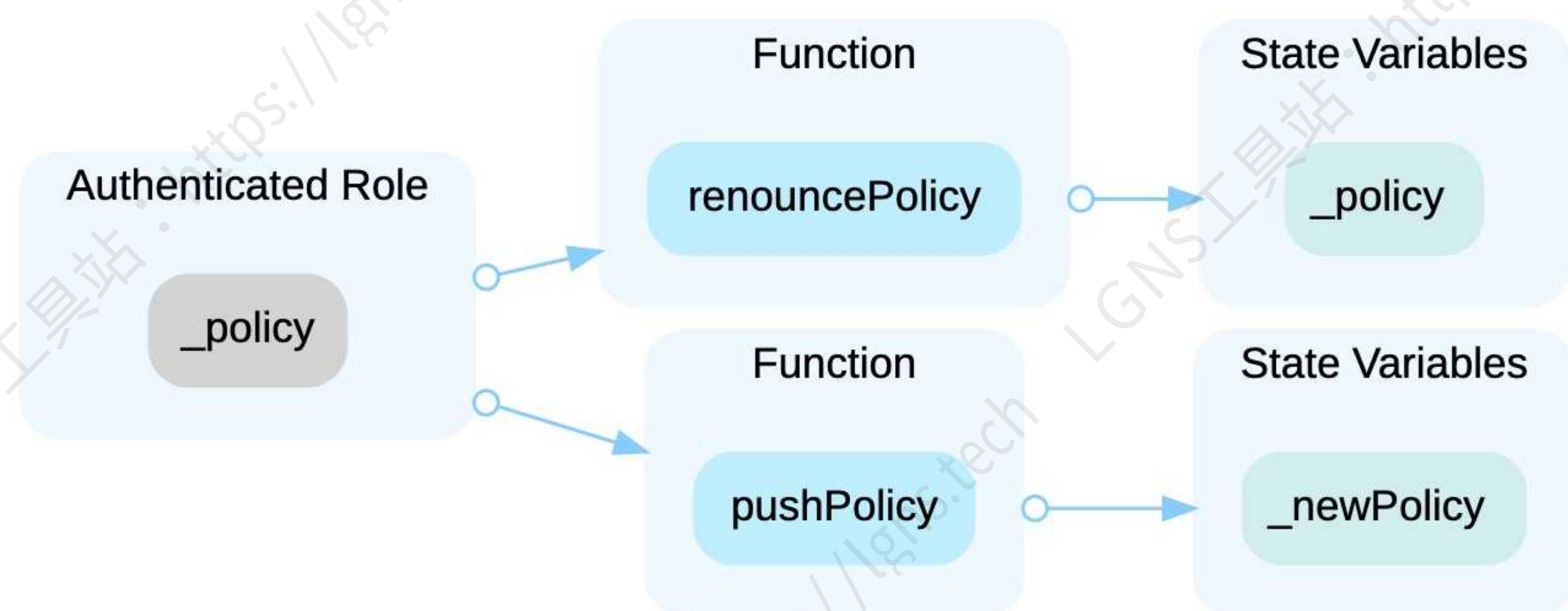
amounts of sOHM out of the staking contract as the LockBonus.



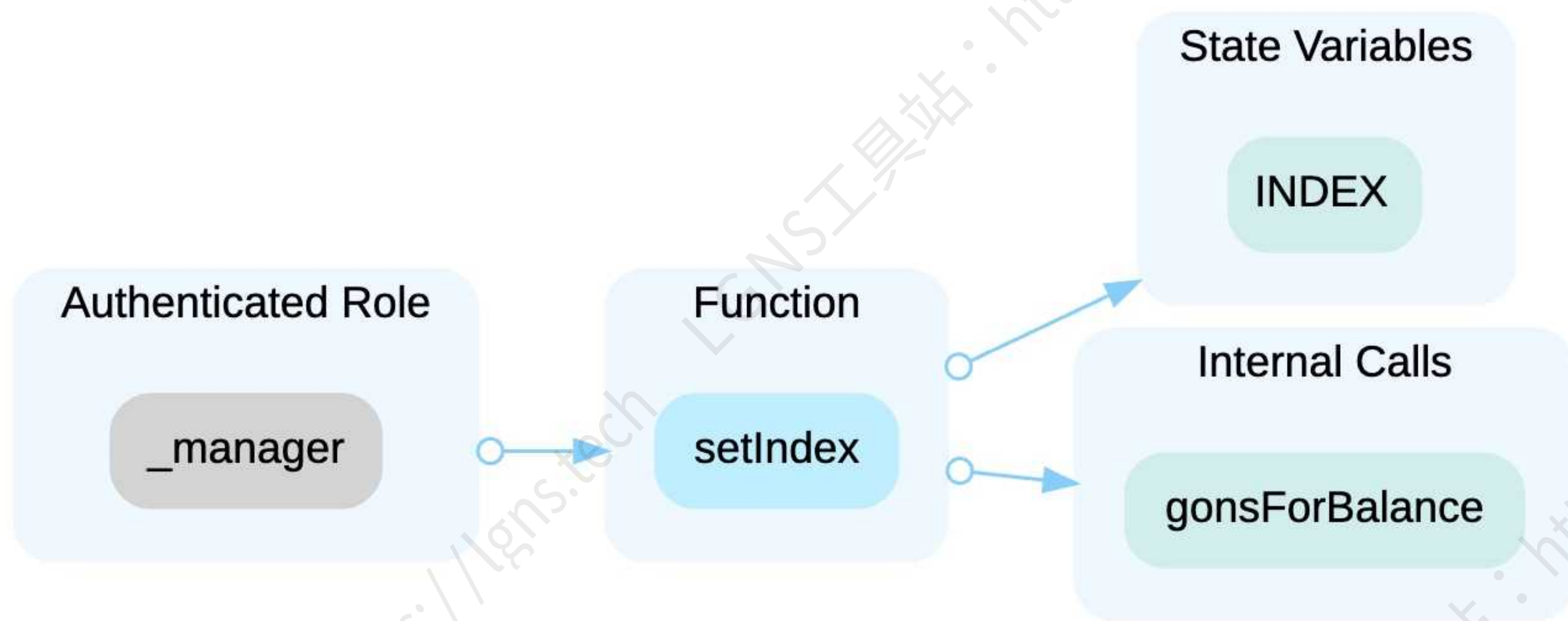
In the contract `StakingDistributor` the role `_policy` has authority over the functions shown in the diagram below. Any compromise to the `_policy` account may allow the hacker to take advantage of this authority to set adjustment parameters and recipient addresses.



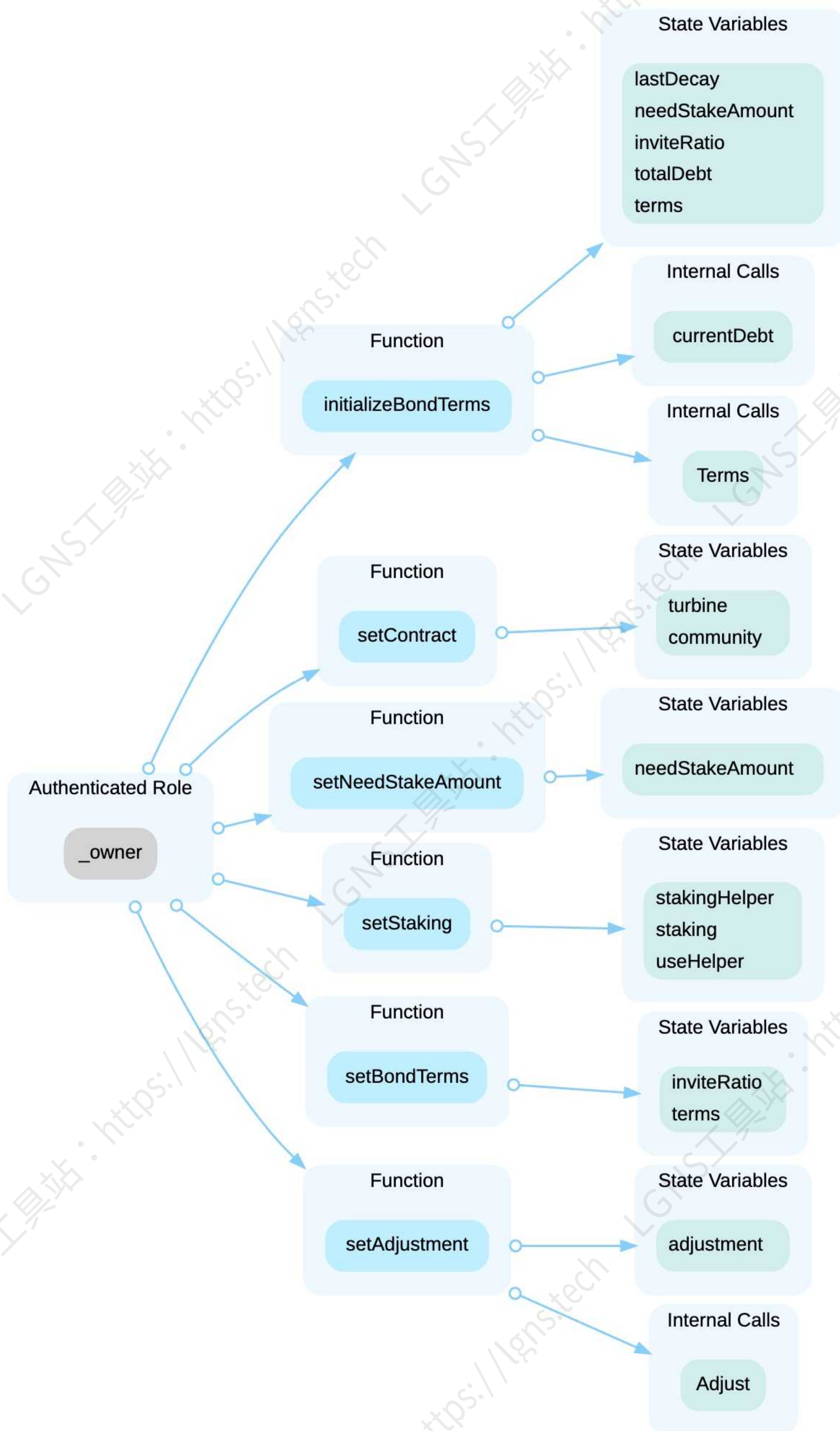
In the contract `StakingDistributor` the role `_policy` has authority over the functions shown in the diagram below. Any compromise to the `_policy` account may allow the hacker to take advantage of this authority and renounce/transfer the policy role.



In the contract `sLGNS` the role `_manager` has authority over the functions shown in the diagram below. Any compromise to the `_manager` account may allow the hacker to take advantage of this authority and modify the index value.

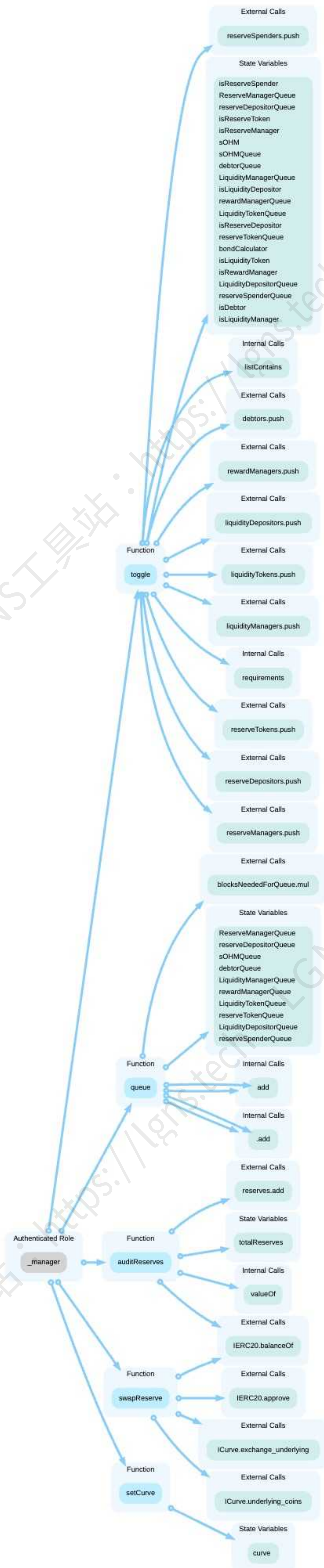


In the contract `WETHBondDepository` the role `_owner` has authority over the functions shown in the diagram below. Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and modify bond settings and term arguments.



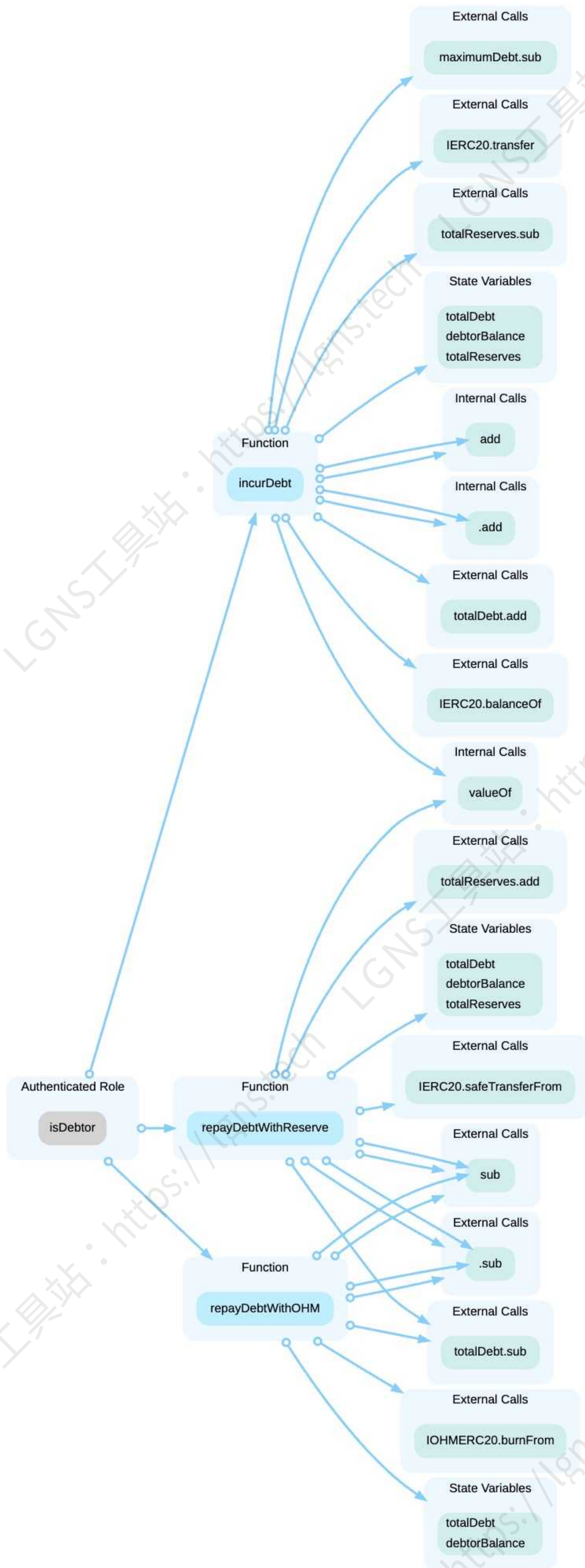
In the contract `Treasury` the role `_manager` has authority over the functions shown in the diagram below. Any compromise to the `_manager` account may allow the hacker to take advantage of this authority and

- modify queue block time for different type of addresses
- verify queue and set boolean in queue mappings
- swap reserve tokens to main coins through `swapReserve`
- modify curve contract address

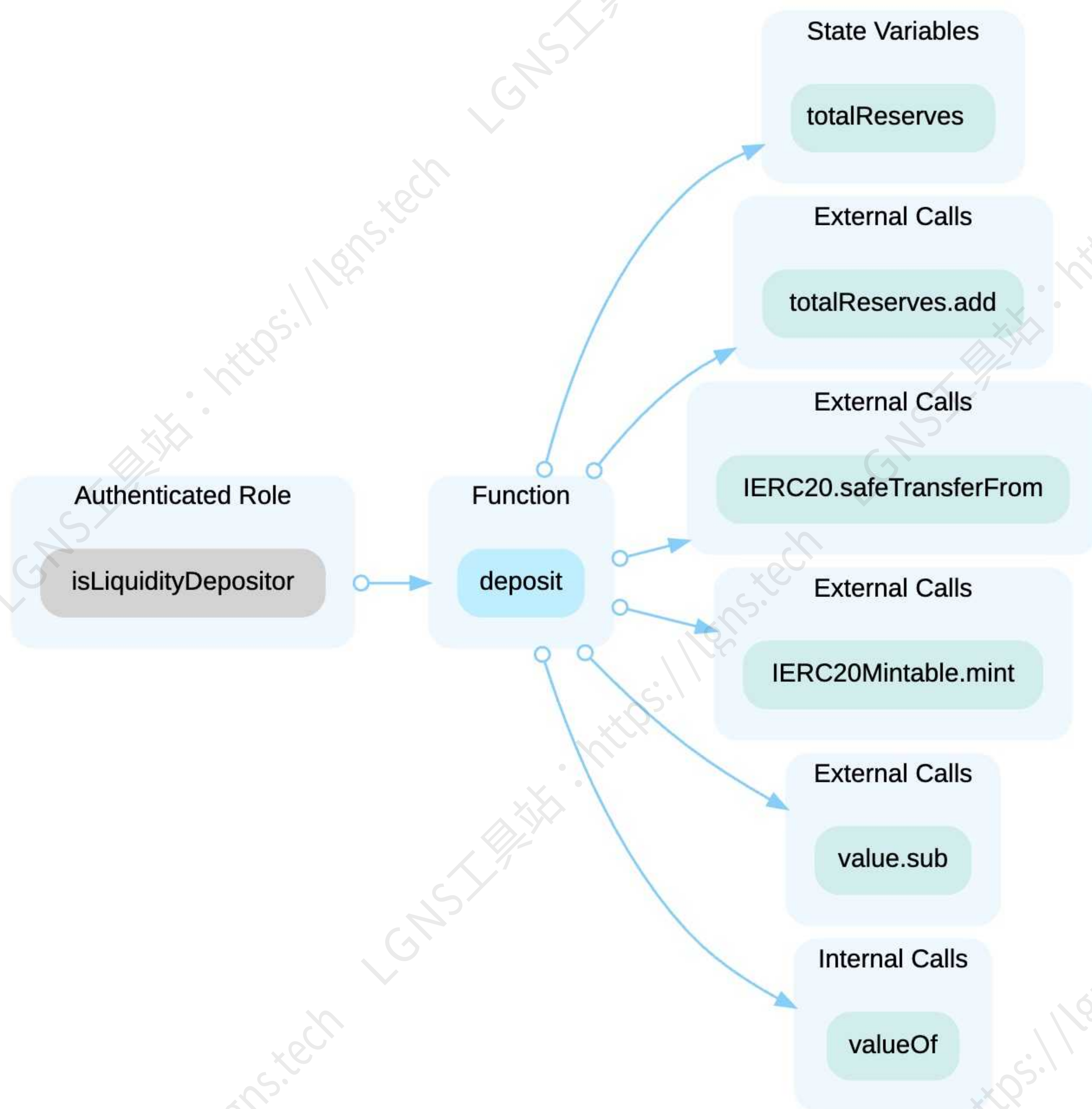


In the contract `Treasury` the role `isDebtor` has authority over the functions shown in the diagram below. Any compromise to the `isDebtor` account may allow the hacker to take advantage of this authority and borrow reserves from the treasury.

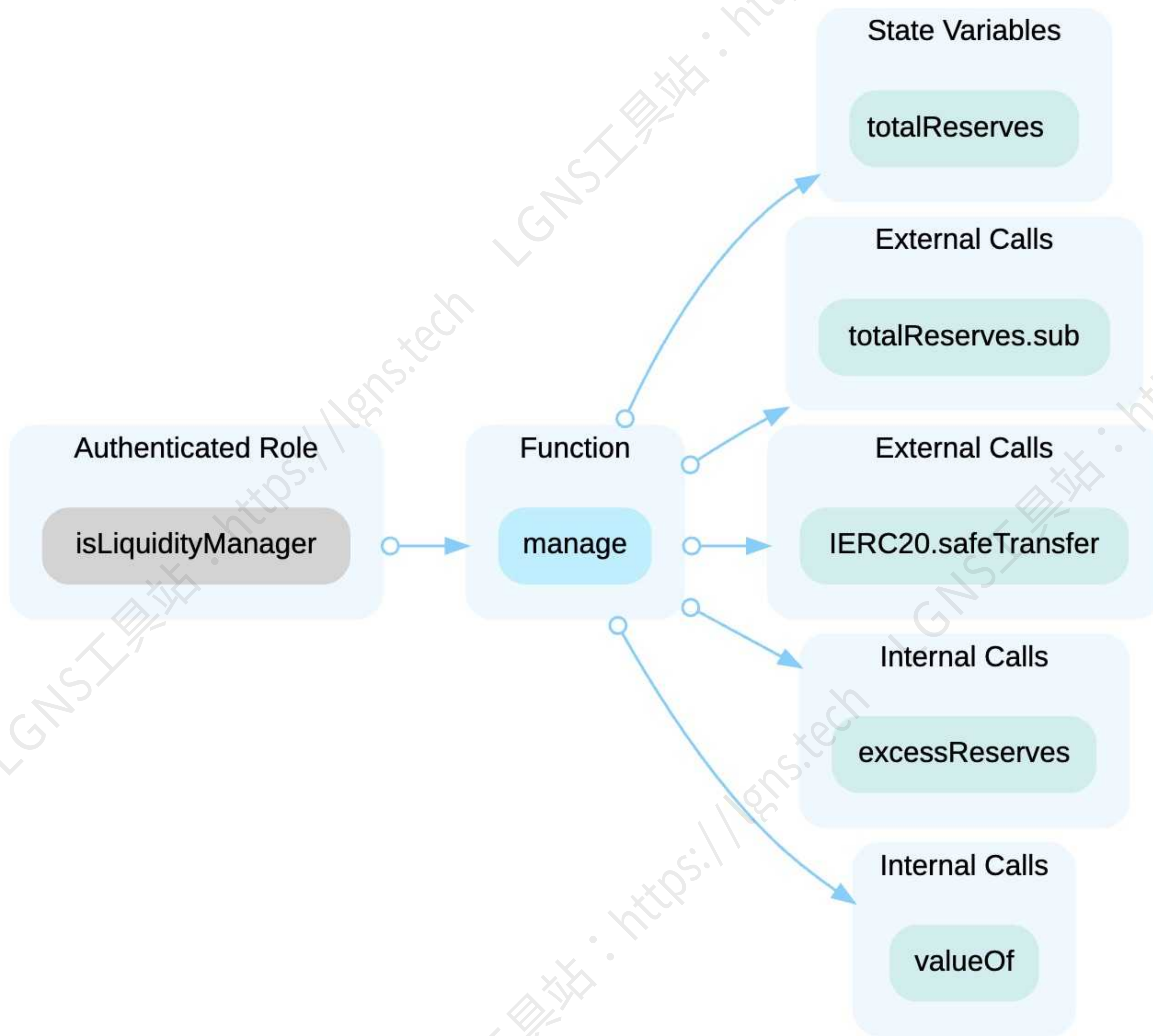
Please note that bad debt might occur if the `Debtor` role is malicious because there is no penalty mechanism.



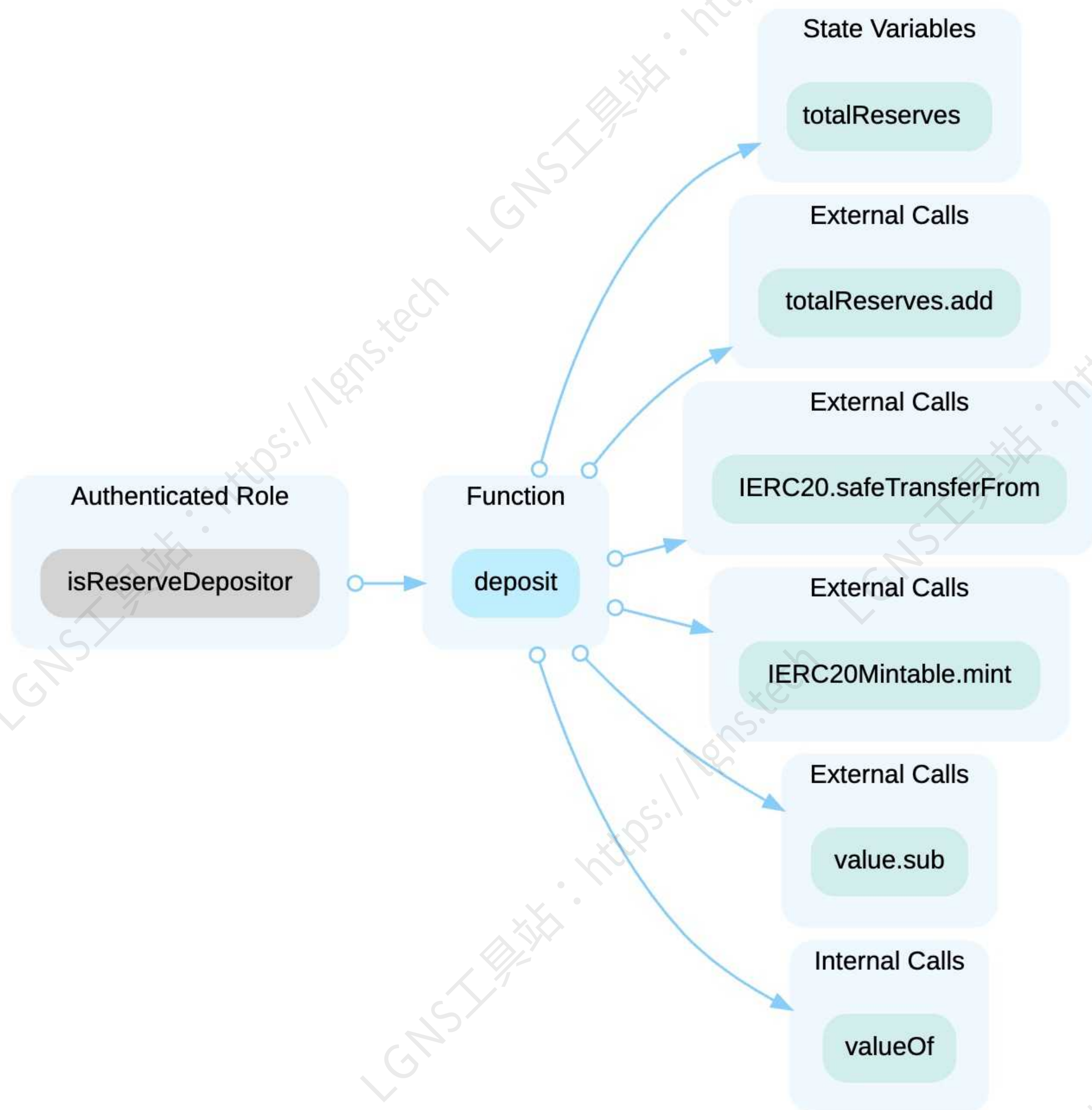
In the contract `Treasury` the role `isLiquidityDepositor` has authority over the functions shown in the diagram below. Any compromise to the `isLiquidityDepositor` account may allow the hacker to take advantage of this authority and deposit liquidity asset to the treasury.



In the contract `Treasury` the role `LiquidityManager` has authority over the functions shown in the diagram below. Any compromise to the `isLiquidityManager` account may allow the hacker to take advantage of this authority and **withdraw** liquidity asset from the treasury through the `manage()` function.

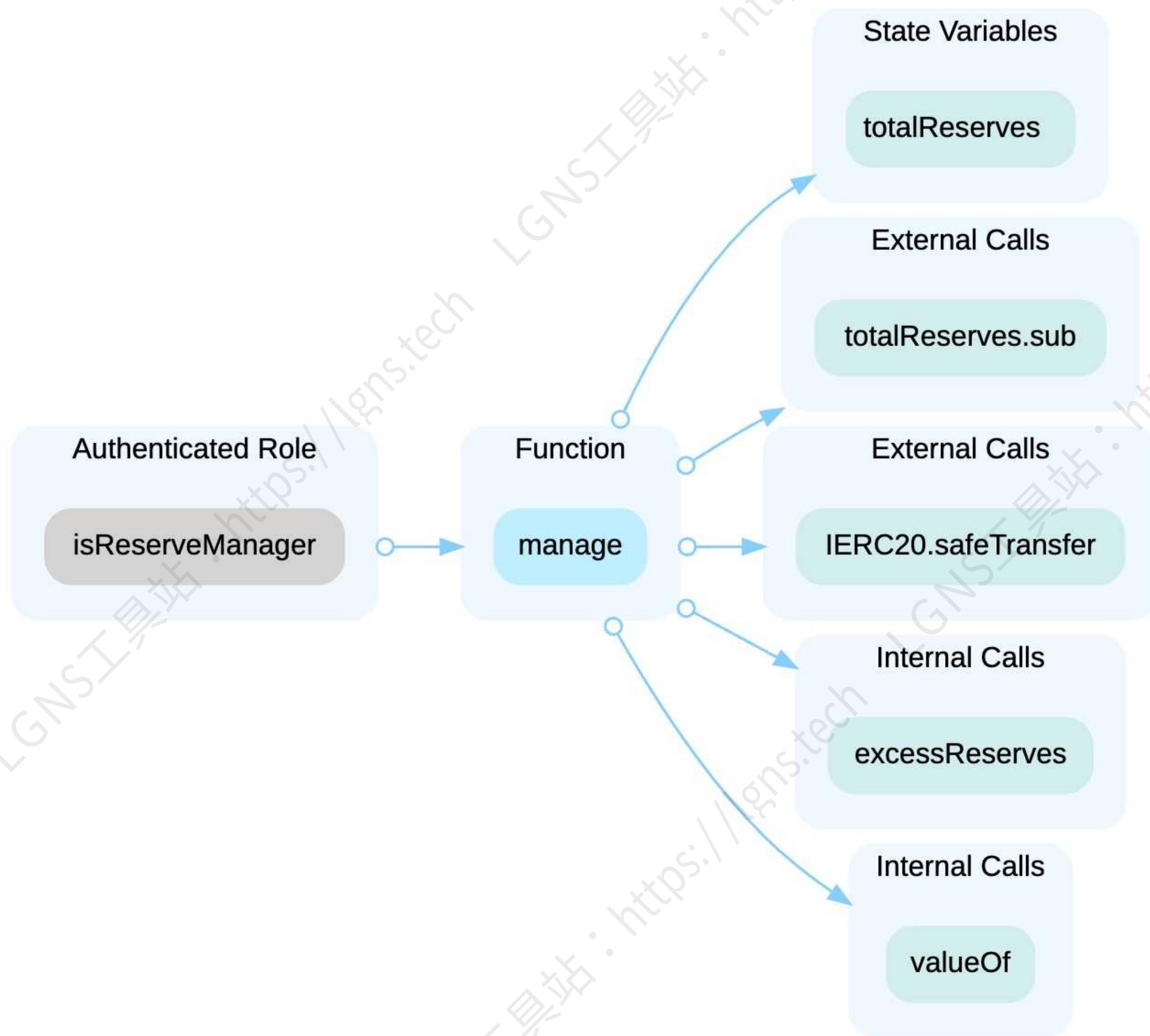


In the contract `Treasury` the role `ReserveDepositor` has authority over the functions shown in the diagram below. Any compromise to the `isReserveDepositor` account may allow the hacker to take advantage of this authority and deposit non-liquidity asset to the treasury.



In the contract `Treasury` the role `ReserveManager` has authority over the functions shown in the diagram below. Any compromise to the `isReserveManager` account may allow the hacker to take advantage of this authority and

- mint OHM tokens to any address through the function `mintRewards()`
- withdraw assets from the treasury through the function `manage()`



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 ($\frac{2}{3}$, $\frac{3}{5}$) 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.

I Alleviation

[ORIGIN Team, 04/09/2024]: After the DeFi system stabilizes, the `_owner`, `_manager`, and other permissions will be transferred to DAO management.

[CertiK, 04/09/2024]: It is suggested to implement the aforementioned methods to avoid centralized failure. Also, it strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

ORG-01 | OWNERSHIP CAN BE REGAINED AFTER RENOUNCEMENT

Category	Severity	Location	Status
Logical Issue	● Major	RedeemHelper.sol (ec22967): 48; Staking.sol (ec22967): 512; StakingDistributor.sol (ec22967): 321; sERC20.sol (ec22967): 984	● Resolved

Description

After ownership is renounced, the potential for the original owner (or the `_policy` role) to regain ownership exists due to the contract code not resetting the state variable for the new owner candidate to a default value.

This situation arises because the function to renounce ownership does not properly clear or update the related state variable. This allows the original owner to exploit this vulnerability under certain conditions, bypassing security checks and regaining ownership of the contract by calling the relevant function again.

Proof of Concept

Foundry Test:

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

import "forge-std/Test.sol";
import "../src/sERC20.sol";

/*
    user1 stake() 10 OHM, epoch.distribute is 0, so sOHM.rebase() is not called
    warmUpContract 10 sOHM
    user1 claim(), user1 ~10 sOHM, circulatingSupply() ~ 10, contractBalance() == 10

    user2 stake() 10 OHM,
*/

contract sERC20Test is Test {
    sLGNS public sToken;
    address public owner = vm.addr(1);

    function setUp() public {
        vm.prank(owner);
        sToken = new sLGNS();
    }

    function testRegainOwnership() public {
        vm.startPrank(owner);

        assert(sToken.manager() != address(0)); // owner is not address(0)

        sToken.pushManagement(owner); // push the ownership to self
        sToken.renounceManagement(); // renounce ownership
        assert(sToken.manager() == address(0)); // owner is address(0)

        sToken.pullManagement(); // pull the ownership
        assert(sToken.manager() != address(0)); // regain the ownership

        vm.stopPrank();
    }
}
```

Recommendation

We recommend modifying codes to avoid regaining ownership after renouncement.

Alleviation

The client fixed the code by adding constraint:

```
1  require( _owner != address(0), "Ownable: must not be renounced");
```

ORN-02 | LOSS OF STAKED TOKENS FOR USERS DUE TO THE BondDepositoryDai AND StakingHelper IMPLEMENTATION

Category	Severity	Location	Status
Logical Issue	Major	1/BondDepository.sol (v2): 1319; 1/StakingHelper.sol (v2): 97	Resolved

Description

In the `BondDepositoryDai` contract, an attempt to stake tokens on behalf of a user inadvertently results in the tokens being staked for the `BondDepositoryDai` contract itself, rather than the intended user. This issue arises from the `StakingHelper` contract's `stake()` function, which interprets `msg.sender` (the `BondDepositoryDai` contract in this context) as the entity for whom tokens are being staked. Consequently, when `BondDepositoryDai` calls `IStakingHelper(stakingHelper).stake(_amount);`, the `StakingHelper` stakes tokens for `BondDepositoryDai` instead of the user, potentially leading to loss of the staked tokens for the users.

```
1319 IStakingHelper(stakingHelper).stake(_amount);
```

```
contract StakingHelper {

    address public immutable staking;
    address public immutable OHM;

    constructor ( address _staking, address _OHM ) {
        require( _staking != address(0) );
        staking = _staking;
        require( _OHM != address(0) );
        OHM = _OHM;
    }

    function stake( uint _amount ) external {
        IERC20( OHM ).transferFrom( msg.sender, address(this), _amount );
        IERC20( OHM ).approve( staking, _amount );
        IStaking( staking ).stake( _amount, msg.sender );
        IStaking( staking ).claim( msg.sender );
    }
}
```

Proof of Concept

Foundry test:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.7.5;
pragma abicoder v2;

import "forge-std/Test.sol";

contract BondDepositoryDaiSimplified {
    address public immutable stakingHelper;

    constructor ( address _stakingHelper) {
        require( _stakingHelper != address(0) );
        stakingHelper = _stakingHelper;
    }

    function stakeOrSend(uint256 _amount) public {
        IStakingHelper(stakingHelper).stake(_amount);
    }
}

contract StakingHelperSimplified {
    address public immutable staking;

    constructor ( address _staking) {
        require( _staking != address(0) );
        staking = _staking;
    }

    function stake( uint256 _amount ) external {
        IStaking( staking ).stake( _amount, msg.sender );
    }
}

contract StakingSimplified {
    mapping(address => uint256) public stakedAmount;

    function stake(uint256 _amount, address _recipient) external {
        stakedAmount[_recipient] += _amount;
    }
}

interface IStakingHelper {
    function stake(uint _amount) external;
}

interface IStaking {
    function stake( uint _amount, address _recipient ) external;
}
```

```
contract StakingHelperTest is Test {
    BondDepositoryDaiSimplified public bondDepository;
    StakingSimplified public staking;
    StakingHelperSimplified public stakingHelper;

    function setUp() public {
        staking = new StakingSimplified();
        stakingHelper = new StakingHelperSimplified(address(staking));
        bondDepository = new BondDepositoryDaiSimplified(address(stakingHelper));
    }

    function testIssue() public {
        address user = vm.addr(1);
        vm.prank(user);
        bondDepository.stakeOrSend(1000); // stake 1000 tokens via the
        `bondDepository` contract

        // check staking amounts
        require(staking.stakedAmount(user) == 0, "error0"); // user has no stake
        require(staking.stakedAmount(address(bondDepository)) == 1000, "error1"); //
        `bondDepository` contract has a stake
    }
}
```

Recommendation

It is recommended to reconsider the design and revise the code.

Alleviation

The team heeded our advice and fixed the issue in the new codebase.

TOR-02 | MANAGER ROLE IN TREASURY COULD TRANSFER OUT TOKEN ASSET IN CONTRACT WITH A MALICIOUS `curve` ADDRESS

Category	Severity	Location	Status
Centralization	● Major	Treasury.sol (ec22967): 846~848	● Acknowledged

Description

The treasury contract contains a function, `swapReserve`, which approves a specified amount of tokens to a designated curve address for asset exchange. However, the contract also exposes a `setCurve` function that allows a manager to change the curve address. If the curve address is altered to point to a malicious contract, it enables an attacker (with manager privileges) to redirect all assets within the treasury to the attacker's control, exploiting the approval given to the malicious curve address.

```
824 function swapReserve(  
825     uint256 i,  
826     uint256 j,  
827     // address _fromToken,  
828     // address _toToken,  
829     uint _amount  
830 ) external onlyManager {  
831     address fromToken = ICurve(curve).underlying_coins(i);  
832     address toToken = ICurve(curve).underlying_coins(j);  
833     require(isReserveToken[fromToken], "Not accepted");  
834     require(isReserveToken[toToken], "Not accepted");  
835     require(  
836         IERC20(fromToken).balanceOf(address(this)) >= _amount,  
837         "Insufficient reserves"  
838     );  
839  
840     IERC20(fromToken).approve(curve, _amount);  
841     ICurve(curve).exchange_underlying(i, j, _amount, 1);  
842  
843     emit ReservesManaged(fromToken, _amount);  
844 }  
845  
846 function setCurve(address _curve) external onlyManager {  
847     curve = _curve;  
848 }  
849 }  
850
```

Besides, the function `manage` also allows manager role to withdraw assets from the treasury, which impose centralization

concern and risk of rug-pull.

```
520 function manage(address _token, uint _amount) external {
521     if (isLiquidityToken[_token]) {
522         require(isLiquidityManager[msg.sender], "Not approved");
523     } else {
524         require(isReserveManager[msg.sender], "Not approved");
525     }
526
527     uint value = valueOf(_token, _amount);
528     require(value <= excessReserves(), "Insufficient reserves");
529
530     totalReserves = totalReserves.sub(value);
531     emit ReservesUpdated(totalReserves);
532
533     IERC20(_token).safeTransfer(msg.sender, _amount);
534
535     emit ReservesManaged(_token, _amount);
536 }
```

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 ($\frac{2}{3}$, $\frac{3}{5}$) 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.

I Alleviation

[ORIGIN Team, 04/09/2024]: After the DeFi system stabilizes, the `_owner`, `_manager`, and other permissions will be transferred to DAO management.

BDO-01 | RISK OF BOND PRICE MANIPULATION DUE TO UNCONTROLLED `setAdjustment()` CALLS

Category	Severity	Location	Status
Logical Issue	● Medium	BondDepository.sol (ec22967): 1024~1042	● Resolved

Description

There is no upper limit restricting parameter of `setAdjustment()`, potentially enabling arbitrary increments on control variables. The commented-out requirement line suggests an intended check against the `terms.controlVariable` to restrict the increment size, which is important for maintaining controlled adjustments.

```
1024     function setAdjustment(  
1025         bool _addition,  
1026         uint _increment,  
1027         uint _target,  
1028         uint _buffer  
1029     ) external onlyPolicy {  
1030         // require(  
1031             //     _increment <= terms.controlVariable.mul(25).div(1000),  
1032             //     "Increment too large"  
1033         // );  
1034         adjustment = Adjust({  
1035             add: _addition,  
1036             rate: _increment,  
1037             target: _target,  
1038             buffer: _buffer,  
1039             lastBlock: block.number  
1040         });  
1041     }  
1042 }
```

Besides, the `BondDepository.sol` contract's design permits the potential for bond price manipulation through the unregulated adjustment of the `terms.controlVariable` via the `adjust()` function. This function dynamically alters the `controlVariable` based on the `adjustment.add` boolean flag—incrementing it when `TRUE` and decrementing it when `FALSE`.

Here are the code blocks that related to the bond price calculation:

```
function deposit(
    uint _amount,
    uint _maxPrice,
    address _depositor
) external returns ( uint ) {
    require( _depositor != address(0), "Invalid address" );

    decayDebt();
    .....

    uint value = ITreasury( treasury ).valueOf( principle, _amount );
    uint payout = payoutFor( value );

    .....
    adjust(); /**COMMENT from CertiK: This function is used to modify the value of
variable terms.controlVariable according to the value of adjustment.add. When
adjustment.add is TRUE, terms.controlVariable increases each time, otherwise
terms.controlVariable decreases each time.**
```

```
function adjust() internal {
    uint blockCanAdjust = adjustment.lastBlock.add( adjustment.buffer );
    if(adjustment.rate != 0 && block.number >= blockCanAdjust) {
        uint initial = terms.controlVariable;
        if (adjustment.add) {
            terms.controlVariable = terms.controlVariable.add( adjustment.rate );
            if (terms.controlVariable >= adjustment.target) {
                adjustment.rate = 0;
            }
        } else {
            terms.controlVariable = terms.controlVariable.sub(adjustment.rate);
            if (terms.controlVariable <= adjustment.target) {
                adjustment.rate = 0;
            }
        }
        adjustment.lastBlock = block.number;
        emit ControlVariableAdjustment(initial, terms.controlVariable, adjustment.rate,
adjustment.add);
    }
}
```

```
function decayDebt() internal {
    totalDebt = totalDebt.sub( debtDecay() );
    lastDecay = block.number;
}

function payoutFor( uint _value ) public view returns ( uint ) {
    return FixedPoint.fraction( _value, bondPrice() ).decode112with18().div( 1e16 );
}

function bondPrice() public view returns ( uint price_ ) {
    price_ = terms.controlVariable.mul(debtRatio()).add(10000000000).div( 1e7 );
    if ( price_ < terms.minimumPrice ) {
        price_ = terms.minimumPrice;
    }
}

function debtRatio() public view returns ( uint debtRatio_ ) {
    uint supply = IERC20(OHM).totalSupply();
    debtRatio_ = FixedPoint.fraction(
        currentDebt().mul(1e9),
        supply
    ).decode112with18().div(1e18);
}

function currentDebt() public view returns (uint) {
    return totalDebt.sub(debtDecay());
}

function debtDecay() public view returns (uint decay_) {
    uint blocksSinceLast = block.number.sub(lastDecay);
    decay_ = totalDebt.mul(blocksSinceLast).div(terms.vestingTerm);
    if (decay_ > totalDebt) {
        decay_ = totalDebt;
    }
}
```

In the Treasury.sol contract:

```
function valueOf(address _token, uint _amount) public view returns (uint value_) {
    if (isReserveToken[_token]) {
        // convert amount to match LMDA decimals
        value_ =
        _amount.mul(10**IERC20(LMDA).decimals()).div(10**IERC20(_token).decimals());
    } else if (isLiquidityToken[_token]) {
        value_ = IBondCalculator(bondCalculator[_token]).valuation(_token, _amount);
    }
}
```

Referring to the above code, the bond price can be expressed as the following formulas:

$$\text{bondAmount} = \frac{\text{Value}}{\text{bondPrice} \times 10^{16}} \quad (1)$$

$$\text{bondPrice} = \text{controlVariable} \times \frac{\text{totalDebt}'}{\text{totalSupply} \times 10^9} + 100 \quad (2)$$

$$\text{totalDebt}' = \text{totalDebt} \times \left(1 - \frac{\Delta\text{block}}{\text{vestingTerm}}\right) \quad (3)$$

where $\Delta\text{block} \leq \text{vestingTerm}$

- *Value* : the amount of reserve tokens paid by users
- *controlVariable* : `terms.controlVariable` whose rise and fall is controlled by function `adjust()`
- Δblock : the current `block.number` minus the `block.number` at the last call
- *vestingTerm* : the total number of blocks for bond issuance set by the `owner`
- *totalDebt* : record the total issuance of bonds
- *totalDebt'* : record the latest total issuance of bonds
- *totalSupply* : record the total issuance of `OHM` token

Consequently, the bond price formula, which heavily relies on the `controlVariable`, is subject to fluctuation by external factors (struct adjustment) that could be intentionally influenced, affecting the integrity of bond pricing. The lack of checks and balances on the frequency and magnitude of these adjustments can lead to scenarios where the bond prices can be artificially inflated or deflated, depending on the direction of the adjustment and the current market dynamics.

Recommendation

We recommend setting a proper restrictions on adjustment operation to ensure the bond price cannot be manipulated.

Alleviation

The team heeded our advice and fixed the issue in the new codebase.

BDO-05 | MISCALCULATION OF PROFIT DUE TO INCORRECT `_inviteProfit` HANDLING

Category	Severity	Location	Status
Logical Issue	● Medium	BondDepository.sol (ec22967): 1131	● Resolved

Description

In the current implementation of function `deposit`, the profit is mistakenly calculated by `profit = value.sub(payout).sub(fee).add(_inviteProfit)`. The profit is later calculated in `treasury.deposit()` to decide how many OHM tokens will be minted to the `BondDepository` contract.

```
1162     ITreasury(treasury).deposit(_amount, principle, profit.add(_inviteProfit));
```

In treasury.sol:

```
422     uint value = valueOf(_token, _amount);
423     // mint OHM needed and store amount of rewards for distribution
424     send_ = value.sub(_profit);
425     IERC20Mintable(OHM).mint(msg.sender, send_);
```

Auditor believes the `_inviteProfit` should be deducted from the profit instead of adding to the profit. Since this part of OHM tokens will be automatically sent to the `turbine` address during the `redeemForInviteBond` process, miscalculation of profit may lead to failure of user redeem operation due to an insufficient OHM balance in the `bondDepository` contract.

Recommendation

It's recommended that the team correct the calculation of profit by using `ITreasury(treasury).deposit(_amount, principle, profit.sub(_inviteProfit));`.

Alleviation

The team heeded our advice and fixed the issue in the new codebase.

CKP-01 | INCOMPATIBILITY WITH DEFLATIONARY TOKENS

Category	Severity	Location	Status
Logical Issue	Medium	1/BondDepository.sol (v2): 1140; 1/Staking.sol (v2): 601, 657; 1/StakingHelper.sol (v2): 95; BondDepository.sol (ec22967): 1138; wETHBondDepository.sol (ec22967): 1147; Treasury.sol (ec22967): 414	Acknowledged

Description

When transferring deflationary ERC20 tokens, the input amount may not be equal to the received amount due to the charged transaction fee. For example, if a user sends 100 deflationary tokens (with a 10% transaction fee), only 90 tokens actually arrive to the contract. However, a failure to discount such fees may allow the same user to withdraw 100 tokens from the contract, which causes the contract to lose 10 tokens in such a transaction.

In BondDepository.sol

```
1138 IERC20(principle).safeTransferFrom(msg.sender, address(this), _amount);
1139 IERC20(principle).approve(address(treasury), _amount);
```

In Treasury.sol:

```
414 IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
415
416 uint value = valueOf(_token, _amount);
```

Proof of Concept

Foundry test:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.7.5;
pragma abicoder v2;

import "forge-std/Test.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract DeflationaryToken is ERC20 {
    address deadAddress = 0x0000000000000000000000000000000000000000dEaD;
    constructor () ERC20 ("Test", "TEST") {
        _mint(msg.sender, 1000000);
    }

    function _transfer(address sender, address recipient, uint256 amount) internal
    override {
        uint256 burnAmount = 10 * amount / 100;
        uint256 transferAmount = amount - burnAmount;
        super._transfer(sender, deadAddress, burnAmount);
        super._transfer(sender, recipient, transferAmount);
    }
}

contract VictimContract {
    address public token;
    mapping(address => uint256) public stakedAmount;

    constructor (address _token) {
        token = _token;
    }

    function stake(uint256 amount) public {
        ERC20(token).transferFrom(msg.sender, address(this), amount);
        stakedAmount[msg.sender] += amount;
    }

    function unstake() public {
        uint256 amount = stakedAmount[msg.sender];
        ERC20(token).transfer(msg.sender, amount);
        stakedAmount[msg.sender] = 0;
    }
}

contract DeflationaryTokenTest is Test {
    DeflationaryToken public deflationaryToken;
    VictimContract public victimContract;
    address public user = vm.addr(1);

    function setUp() public {
        deflationaryToken = new DeflationaryToken();
    }
}
```

```
victimContract = new VictimContract(address(deflationaryToken));
deflationaryToken.transfer(address(victimContract), 10000);
deflationaryToken.transfer(user, 10000);
}

function testIssue() public {
    vm.startPrank(user);
    deflationaryToken.approve(address(victimContract), 1000);

    uint256 victimBalanceBefore =
deflationaryToken.balanceOf(address(victimContract)); // balance before staking
    victimContract.stake(1000);
    victimContract.unstake();

    uint256 victimBalanceAfter =
deflationaryToken.balanceOf(address(victimContract)); // balance after unstaking

    vm.stopPrank();

    require(victimBalanceAfter < victimBalanceBefore, "error"); // victim
contract lost tokens
}
}
```

Recommendation

We advise the client to regulate the set of tokens supported and add necessary mitigation mechanisms to keep track of accurate balances if there is a need to support non-standard ERC20 tokens in `BondDepository` and `Treasury` contracts.

Alleviation

[ORIGIN Team, 04/09/2024]: The implementation of the program code is to meet business requirements.

ERC-02 | NO UPPER LIMITS FOR `feeRatio` AND `buyFeeRatio`

Category	Severity	Location	Status
Logical Issue	● Medium	ERC20.sol (ec22967): 482, 484	● Acknowledged

Description

The default admin role has the ability to call the `setRatio()` function to adjust the `feeRatio` and `buyFeeRatio` of the ORIGIN ERC-20 token, with no maximum limit specified. In extreme scenarios, these rates could be set to as high as 100%, effectively meaning the receiver would not receive any tokens from buying or selling activities.

Proof of Concept

Foundry test:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.7.5;
pragma abicoder v2;

import "forge-std/Test.sol";

contract OriginWithLotterySimplified {
    uint256 public constant PRECISION = 100 * 1e3;
    uint256 public feeRatio;
    uint256 public buyFeeRatio;

    function setRatio(uint8 ratioType,uint256 ratio) external {
        require(ratio <= PRECISION, "Exceeds precision");
        if(ratioType ==0){
            buyFeeRatio = ratio;
        } else {
            feeRatio = ratio;
        }
    }
}

contract SomeTest is Test {
    OriginWithLotterySimplified public originERC20;

    function setUp() public {
        originERC20 = new OriginWithLotterySimplified();
    }

    function OriginTest() public {
        originERC20.setRatio(0, 100 * 1e3);
        originERC20.setRatio(1, 100 * 1e3);

        require(originERC20.feeRatio() == originERC20.PRECISION(), "error0"); //
        `feeRatio` has been set to 100%
        require(originERC20.buyFeeRatio() == originERC20.PRECISION(), "error1"); //
        `buyFeeRatio` has been set to 100%
    }
}
```

Recommendation

We recommend setting upper limits for `feeRatio` and `buyFeeRatio`.

Alleviation

[CertiK, 04/10/2024]: The updated code changed the require statement from `require(ratio <= PRECISION)` to `require(ratio < PRECISION)`, permitting fee rates to approach 100%. This modification does not resolve the issue.

ETH-01 | MISSING STALENESS CHECKS FOR CHAINLINK ORACLE

Category	Severity	Location	Status
Logical Issue	● Medium	WETHBondDepository.sol (ec22967): 1406	● Resolved

Description

The `assetPrice()` function in the `WETHBondDepository` contract makes use of the Chainlink's `latestRoundData()` function which does not check whether the return value contains outdated data. Consequently, this could result in `assetPrice()` providing an outdated price.

```
1402  /**
1403   * @notice get asset price from chainlink
1404   */
1405  function assetPrice() public view returns (int) {
1406      (, int price, , , ) = priceFeed.latestRoundData();
1407      return price;
1408  }
```

Recommendation

A proposed mitigation checking for valid returned values could be the following:

```
(uint80 roundID, int256 price, , uint256 updatedAt, uint80 answeredInRound) =
priceFeed.latestRoundData();
require(answeredInRound >= roundID, "Stale price");
require(price > 0, "Invalid price");
require(block.timestamp <= updatedAt + stalePriceDelay, "Stale price");
```

`stalePriceDelay` should be set to a reasonable value.

Alleviation

The team heeded our advice and fixed the issue in the new codebase.

ORG-04 | ANYONE CAN CALL `claim()` / `redeem()` FOR ANY ARBITRARY `_recipient`

Category	Severity	Location	Status
Logical Issue	● Medium	BondDepository.sol (ec22967): 1213; Staking.sol (ec22967): 622; wETHBondDepository.sol (ec22967): 1204	● Acknowledged

Description

The public `claim()` function and `redeem()` function in the linked contracts accept an arbitrary `_recipient` address parameter, rather than using `msg.sender`. This design allows any user to call `claim()` / `redeem()` for another account without the account holder's consent. While this could be by design, to facilitate claims through the `StakingHelper` contract on behalf of users, it also opens the possibility for unauthorized claims on other users' behalf. We would like to confirm if this behavior is intentional.

Proof of Concept

Foundry test:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.7.5;
pragma abicoder v2;

import "forge-std/Test.sol";

contract StakingSimplified {
    mapping(address => uint256) public stakedAmount;

    function stake(uint256 _amount, address _recipient) external {
        stakedAmount[_recipient] += _amount;
    }

    function claim(address _recipient) external {
        stakedAmount[_recipient] = 0;
    }
}

contract SomeTest is Test {
    StakingSimplified public staking;

    function setUp() public {
        staking = new StakingSimplified();
    }

    function testIssue() public {
        address user1 = vm.addr(1);
        address user2 = vm.addr(2);

        vm.prank(user1);
        staking.stake(100, user1);
        require(staking.stakedAmount(user1) == 100, "error0"); // user1 stakes

        vm.prank(user2);
        staking.claim(user1);
        require(staking.stakedAmount(user1) == 0, "error1"); // user2 claims on
        behalf of user1
    }
}
```

Recommendation

It's recommended that implement proper access control mechanism to prevent unauthorized claim and redeem operations.

Alleviation

[ORIGIN Team, 04/09/2024]: The implementation of the program code is to meet business requirements.

ORG-08 | OUT-OF-SCOPE DEPENDENCIES

Category	Severity	Location	Status
Volatile Code	Medium	BondDepository.sol (ec22967): 1142, 1305; ERC20.sol (ec22967): 520	Acknowledged

Description

The linked contracts are serving as the underlying entity to interact with the out-of-scope contracts

`ICommunity(community)`, `ITurbine(turbine)` and `IFeeReceiver(feeReceiver)`.

The scope of the audit treats these entities as black boxes and assumes their functional correctness. However, in the real world, external dependencies and out-of-scope contracts can be compromised and this may lead to lost or stolen assets.

In BondDepository.sol:

```
1142 address inviteAddress = ICommunity(community).referrerOf(_depositor);
```

```
1305 ITurbine(turbine).receiveTurbine(_recipient, _amount);
```

In ERC20.sol:

```
520 IFeeReceiver(feeReceiver).triggerSwapFeeForLottery(fee);
```

Recommendation

We recommend that the project team constantly monitor the functionality of out-of-scope contracts and dependencies to mitigate any side effects that may occur when unexpected changes are introduced.

Alleviation

[ORIGIN Team, 04/09/2024]: The implementation of the program code is to meet business requirements.

ORN-01 | COMPILATION ERRORS IN WETHBondDepository AND BondDepositoryDai CONTRACTS

Category	Severity	Location	Status
Coding Issue	● Medium	1/BondDepository.sol (v2); 1/wETHBondDepository.sol (v2)	● Resolved

Description

The `WETHBondDepository` contract fails to compile due to an "Undeclared identifier: stalePriceDelay" error, indicating a missing declaration for `stalePriceDelay`. Similarly, the `BondDepositoryDai` contract encounters a "Wrong argument count for function call" error in the invocation of `IStakingHelper(stakingHelper).stake(_amount)`, suggesting a discrepancy between expected and provided parameters in the `stake` function call. These issues prevent successful compilation and, consequently, deployment of the contracts.

Recommendation

It is recommended to address these compilation errors to ensure the contracts can be deployed.

Alleviation

The team heeded our advice and fixed the issue in the new codebase.

ORN-03 | INCORRECT StakingHelper CONTRACT VERSION USED IN BondDepositoryDai AND WETHBondDepository

Category	Severity	Location	Status
Design Issue	Medium	1/BondDepository.sol (v2): 1316; 1/StakingHelper.sol (v2): 94; 1/WETHBondDepository.sol (v2): 1303	Resolved

Description

The `stakeOrSend()` function in the `BondDepositoryDai` and `WETHBondDepository` contracts is designed to call the `IStakingHelper(stakingHelper).stake(_amount, _recipient)` function in some cases, which requires two parameters. However, the `stake()` function within the reviewed `StakingHelper` contract accepts only one parameter. This discrepancy raises concerns that the incorrect contract version might be in use.

```
82 contract StakingHelper {
83
84     address public immutable staking;
85     address public immutable OHM;
86
87     constructor ( address _staking, address _OHM ) {
88         require( _staking != address(0) );
89         staking = _staking;
90         require( _OHM != address(0) );
91         OHM = _OHM;
92     }
93
94     function stake( uint _amount ) external {
95         IERC20( OHM ).transferFrom( msg.sender, address(this), _amount );
96         IERC20( OHM ).approve( staking, _amount );
97         IStaking( staking ).stake( _amount, msg.sender );
98         IStaking( staking ).claim( msg.sender );
99     }
100 }
```

Proof of Concept

Foundry test:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.7.5;
pragma abicoder v2;

import "forge-std/Test.sol";

contract BondDepositoryDaiSimplified {
    address public immutable stakingHelper;

    constructor ( address _stakingHelper) {
        require( _stakingHelper != address(0) );
        stakingHelper = _stakingHelper;
    }

    function stakeOrSend(uint256 _amount) public {
        IStakingHelper(stakingHelper).stake(_amount, msg.sender);
    }
}

contract StakingHelperSimplified {
    address public immutable staking;

    constructor ( address _staking) {
        require( _staking != address(0) );
        staking = _staking;
    }

    function stake( uint256 _amount ) external {
        IStaking( staking ).stake( _amount, msg.sender );
    }
}

contract StakingSimplified {
    mapping(address => uint256) public stakedAmount;

    function stake(uint256 _amount, address _recipient) external {
        stakedAmount[_recipient] += _amount;
    }
}

interface IStakingHelper {
    function stake(uint _amount, address _recipient) external;
}

interface IStaking {
    function stake( uint _amount, address _recipient ) external;
}
```

```
contract StakingHelperV2Test is Test {
    BondDepositoryDaiSimplified public bondDepository;
    StakingSimplified public staking;
    StakingHelperSimplified public stakingHelper;

    function setUp() public {
        staking = new StakingSimplified();
        stakingHelper = new StakingHelperSimplified(address(staking));
        bondDepository = new BondDepositoryDaiSimplified(address(stakingHelper));
    }

    function testIssue() public {
        address user = vm.addr(1);
        vm.prank(user);
        vm.expectRevert(); // the following function call reverts
        bondDepository.stakeOrSend(1000);
    }
}
```

Recommendation

It's recommended that the team resolve the discrepancy between the linked code.

Alleviation

The team heeded our advice and fixed the issue in the new codebase.

SDO-01 | INCORRECT MINTING AMOUNTS DUE TO DYNAMIC `totalSupply`

Category	Severity	Location	Status
Logical Issue	● Medium	StakingDistributor.sol (ec22967): 389	● Acknowledged

Description

The `distribute()` function in the `Distributor` contract calculates minting amounts for OHM tokens based on the current `totalSupply`, applying a rate (`_rate`) to determine the reward for each recipient. Since `totalSupply` increases with each minting operation within the same distribution cycle, recipients listed later in the iteration receive more rewards due to the inflated supply. This sequential dependency can lead to unintended distribution of rewards and potentially exceed predetermined minting limits.

```
function distribute() external returns ( bool ) {
    if ( nextEpochBlock <= block.number ) {
        nextEpochBlock = nextEpochBlock.add( epochLength ); // set next epoch
    }

    // distribute rewards to each recipient
    for ( uint i = 0; i < info.length; i++ ) {
        if ( info[ i ].rate > 0 ) {
            ITreasury( treasury ).mintRewards( // mint and send from
                info[ i ].recipient,
                nextRewardAt( info[ i ].rate )
            );
            adjust( i ); // check for adjustment
        }
    }
    return true;
} else {
    return false;
}
```

```
function nextRewardAt( uint _rate ) public view returns ( uint ) {
    return IERC20( OHM ).totalSupply().mul( _rate ).div( 1000000 );
}
```

Scenario

The `distribute()` function aims to mint tokens equivalent to 10% of the current total supply to four different recipients, with the initial total supply being 100.

- The first recipient receives 10 tokens (totalSupply becomes 110);
- The second recipient receives 11 tokens (totalSupply becomes 121);
- The third recipient receives 12 tokens (totalSupply becomes 133);
- The fourth recipient receives 13 tokens (totalSupply becomes 146);

Proof of Concept

Foundry test:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.7.5;
pragma abicoder v2;

import "forge-std/Test.sol";

contract DistributorSimplified {
    mapping(address => uint256) public rewards;
    address[] public recipients;
    uint256 public supply = 100;

    constructor (address[] memory _recipients) {
        for (uint i = 0; i < _recipients.length; i++) {
            recipients.push(_recipients[i]);
        }
    }

    function distribute() external {
        // distribute rewards to each recipient
        for (uint i = 0; i < recipients.length; i++) {
            uint256 rewardAmount = 10 * supply / 100;
            rewards[recipients[i]] += rewardAmount;
            supply += rewardAmount;
        }
    }
}

contract SomeTest is Test {
    DistributorSimplified public distributor;

    function setUp() public {
        address[] memory _recipients = new address[](4);
        _recipients[0] = vm.addr(1);
        _recipients[1] = vm.addr(2);
        _recipients[2] = vm.addr(3);
        _recipients[3] = vm.addr(4);
        distributor = new DistributorSimplified(_recipients);
    }

    function testIssue() public {
        distributor.distribute();
        // rewards for the recipients are not the same
        require(distributor.rewards(distributor.recipients(0)) == 10, "error 0");
        require(distributor.rewards(distributor.recipients(1)) == 11, "error 1");
        require(distributor.rewards(distributor.recipients(2)) == 12, "error 2");
        require(distributor.rewards(distributor.recipients(3)) == 13, "error 3");
    }
}
```

Recommendation

It is recommended to revise the code to make the linked code logic comply with the intended design.

Alleviation

[ORIGIN Team, 04/19/2024]: There will be only one staking contract to distribute.

TOR-01 | POTENTIAL EXPLOITATION VIA sOHM TRANSFERS AND DEBT MECHANISM IN Treasury CONTRACT

Category	Severity	Location	Status
Design Issue	● Medium	Treasury.sol (ec22967): 458	● Acknowledged

Description

By using the `incurDebt()` function in the `Treasury` contract, an approved debtor is allowed to borrow `_token` as long as they hold `sOHM`, with the maximum borrowing amount being equal to their `sOHM` balance. This process could lead to issues, as `sOHM` can be transferred to another approved debtor. For instance, if user A has 100 `sOHM` and borrows 100 `_token`, A can then transfer the `sOHM` to B, who is then also able to borrow 100 `_token`, and this pattern can continue indefinitely.

Furthermore, a debtor can:

1. Use `Treasury.incurDebt()` to borrow `_token`.
2. Then use `Treasury.deposit()` to deposit these `_token` and receive `OHM` tokens.
3. Subsequently, use `Staking.stake()` to stake these `OHM` tokens and `Staking.claim()` to acquire `sOHM` tokens.
4. As the debtor's `sOHM` balance increases, they could repeat this cycle to accumulate more tokens, exacerbating the potential for loss.

Recommendation

We would like to understand whether the debtors can be arbitrary users or if they are controlled by the project owner. We recommend reviewing the design and sharing the intended framework with us.

Alleviation

[ORIGIN Team, 04/09/2024]: The implementation of the program code is to meet business requirements.

BDO-02 | `totalDebt` COULD EXCEED `maxDebt`

Category	Severity	Location	Status
Logical Issue	Minor	BondDepository.sol (ec22967): 1115	Resolved

Description

The following require statement checks that `totalDebt` before `deposit` does not exceed `maxDebt`.

```
require( totalDebt <= terms.maxDebt, "Max capacity reached" );
```

However, `totalDebt` is incremented in the same `deposit` function, and `totalDebt + value` could exceed `maxDebt` after the `deposit` function is executed.

```
totalDebt = totalDebt.add( value );
```

Recommendation

We recommend moving the require statement to after the calculation of the variable value and the increment of `totalDebt`.

Alleviation

The team heeded our advice and fixed the issue in the new codebase.

ETH-02 | INCORRECT EVENT PARAMETERS

Category	Severity	Location	Status
Logical Issue	Minor	wETHBondDepository.sol (ec22967): 1234	Resolved

Description

In function `redeem()`, event `BondRedeemed` contains incorrect event parameter. Here the `bondInfo[_recipient].payout` should be `bondInfoData[_recipient][_id].payout`.

```
1234     emit BondRedeemed(_recipient, payout, bondInfo[_recipient].payout);
```

Recommendation

We recommend correcting the event parameters by using `bondInfoData[_recipient][_id].payout` instead of `bondInfo[_recipient].payout`.

Alleviation

The team heeded our advice and fixed the issue in the new codebase.

ORG-05 | SUSCEPTIBLE TO SIGNATURE MALLEABILITY

Category	Severity	Location	Status
Volatile Code	Minor	BondDepository.sol (ec22967): 582; ERC20.sol (ec22967): 347; wETHBondDepository.sol (ec22967): 582	Resolved

Description

The signature malleability is possible within the Elliptic Curve cryptographic system. An Elliptic Curve is symmetric on the X-axis, meaning two points can exist with the same `x` value. In the `r`, `s` and `v` representation this permits us to carefully adjust `s` to produce a second valid signature for the same `r`, thus breaking the assumption that a signature cannot be replayed in what is known as a replay-attack.

```
address signer = ecrecover(_hash, v, r, s);
require(
    signer != address(0) && signer == owner,
    "ZeroSwapPermit: Invalid signature"
);
```

Recommendation

We advise to utilize a `recover(.)` function similar to that of the `ECDSA.sol` implementation of OpenZeppelin.

```
require(uint256(s) <=
0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0, "ECDSA: invalid
signature 's' value");
require(v == 27 || v == 28, "ECDSA: invalid signature 'v' value");
```

Alleviation

The team heeded our advice and fixed the issue in the new codebase.

ORG-06 | POSSIBILITY OF CROSS CHAIN REPLAY ATTACK IN `permit()` FUNCTION

Category	Severity	Location	Status
Volatile Code	Minor	BondDepository.sol (ec22967): 582; ERC20.sol (ec22967): 347; sERC20.sol (ec22967): 909; wETHBondDepository.sol (ec22967): 582	Resolved

Description

The `permit()` function performs the operation of deriving signer address from the signature values of `v`, `r` and `s`. The state variable `DOMAIN_SEPARATOR` that is used to calculate hash has a value of `chainid` that is derived only once in the constructor, which does not change after contract deployment. The issue arises in the event of fork when the cross-chain replay attacks can be executed.

The attack scenario can be thought of as if a fork of Ethereum happens and two different networks have id of for example `1` and `9`. The `chainid` coded in `DOMAIN_SEPARATOR` will be the same on contracts residing in both of the forks. If the `chainid 1` is stored in the contract then the `permit` transaction signed for `chainid 1` will be executable on both of the forks.

Recommendation

We advise to construct the `DOMAIN_SEPARATOR` hash inside the `permit` function so the current `chainid` could be fetched and only the transactions signed for current network could succeed.

Alleviation

The team heeded our advice and fixed the issue in the new codebase.

BDO-04 | DEPRECATED USAGE OF `Counters.sol`

Category	Severity	Location	Status
Logical Issue	● Informational	BondDepository.sol (ec22967): 532~550	● Acknowledged

Description

The linked contracts import and use OpenZeppelin's `Counters` contract. OpenZeppelin has deprecated the usage of the `Counters` contract: <https://github.com/OpenZeppelin/openzeppelin-contracts/issues/4233>

Recommendation

Consider removing the usage of deprecated 3rd party contracts.

CKP-02 | FUNCTION MISSES ZERO ADDRESS VALIDATION

Category	Severity	Location	Status
Logical Issue	● Informational	1/sERC20.sol (v2): 1142, 1155, 1168; 1/sERC20.sol (final): 1145, 1158, 1171; 1/sERC20.sol (final3): 1145, 1158, 1171; 1/sERC20.sol (final2): 1145, 1158, 1171; sERC20.sol (ec22967): 1142, 1155, 1168	● Acknowledged

Description

It is expected that calls of the form `approve(spender, amount)` fail if the address in `spender` is the zero address.

It is expected that calls of the form `transferFrom(from, dest, amount)` fail for transfers from or to the zero address. However, the `transferFrom` function in `sLGNS` can transfer tokens from or to the zero address.

It is expected that calls of the form `transfer(recipient, amount)` fail if the recipient address is the zero address.

Recommendation

Ensure that the `approve` function in contract `sLGNS` reverts if invoked with a `spender` address of zero.

Ensure that all calls of the form `transferFrom(from, dest, amount)` in contract `sLGNS` revert if invoked with the zero address.

Ensure that the `transfer` function in contract `sLGNS` reverts if invoked with a recipient address of zero.

ERC-01 | FUNCTION `_burnFrom()` IS PUBLIC

Category	Severity	Location	Status
Design Issue	● Informational	ERC20.sol (ec22967): 423	● Acknowledged

Description

The `_burnFrom()` function in the `OriginERC20Token` contract is public. However, it makes more sense that this function is internal. We would like to confirm if this is the intended design.

Recommendation

We recommend changing the visibility to `internal`.

Alleviation

[ORIGIN Team, 04/09/2024]: This feature is the intended design.

FORMAL VERIFICATION | ORIGIN

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 ERC-20 Compliance

We verified properties of the public interface of those token contracts that implement the ERC-20 interface. This covers

- Functions `transfer` and `transferFrom` that are widely used for token transfers,
- functions `approve` and `allowance` that enable the owner of an account to delegate a certain subset of her tokens to another account (i.e. to grant an allowance), and
- the functions `balanceOf` and `totalSupply`, which are verified to correctly reflect the internal state of the contract.

The properties that were considered within the scope of this audit are as follows (note that overflow properties were excluded from the verification):

Property Name	Title
erc20-totalsupply-succeed-always	<code>totalSupply</code> Always Succeeds
erc20-approve-succeed-normal	<code>approve</code> Succeeds for Valid Inputs
erc20-approve-false	If <code>approve</code> Returns <code>false</code> , the Contract's State Is Unchanged
erc20-balanceof-correct-value	<code>balanceOf</code> Returns the Correct Value
erc20-allowance-succeed-always	<code>allowance</code> Always Succeeds
erc20-allowance-correct-value	<code>allowance</code> Returns Correct Value
erc20-transferfrom-never-return-false	<code>transferFrom</code> Never Returns <code>false</code>
erc20-transfer-false	If <code>transfer</code> Returns <code>false</code> , the Contract State Is Not Changed
erc20-approve-correct-amount	<code>approve</code> Updates the Approval Mapping Correctly
erc20-transfer-never-return-false	<code>transfer</code> Never Returns <code>false</code>

Property Name	Title
erc20-transferfrom-fail-exceed-balance	<code>transferFrom</code> Fails if the Requested Amount Exceeds the Available Balance
erc20-transferfrom-false	If <code>transferFrom</code> Returns <code>false</code> , the Contract's State Is Unchanged
erc20-transferfrom-revert-zero-argument	<code>transferFrom</code> Fails for Transfers with Zero Address Arguments
erc20-transfer-revert-zero	<code>transfer</code> Prevents Transfers to the Zero Address
erc20-transfer-exceed-balance	<code>transfer</code> Fails if Requested Amount Exceeds Available Balance
erc20-transferfrom-fail-exceed-allowance	<code>transferFrom</code> Fails if the Requested Amount Exceeds the Available Allowance
erc20-totalsupply-change-state	<code>totalSupply</code> Does Not Change the Contract's State
erc20-allowance-change-state	<code>allowance</code> Does Not Change the Contract's State
erc20-balanceof-change-state	<code>balanceOf</code> Does Not Change the Contract's State
erc20-transferfrom-correct-allowance	<code>transferFrom</code> Updated the Allowance Correctly
erc20-transferfrom-correct-amount	<code>transferFrom</code> Transfers the Correct Amount in Transfers
erc20-transfer-correct-amount	<code>transfer</code> Transfers the Correct Amount in Transfers
erc20-approve-revert-zero	<code>approve</code> Prevents Approvals For the Zero Address
erc20-approve-never-return-false	<code>approve</code> Never Returns <code>false</code>
erc20-balanceof-succeed-always	<code>balanceOf</code> Always Succeeds
erc20-totalsupply-correct-value	<code>totalSupply</code> Returns the Value of the Corresponding State Variable

Verification Results

In the remainder of this section, we list all contracts where formal verification of at least one property was not successful.

There are several reasons why this could happen:

- False: The property is violated by the project.
- Inconclusive: The proof engine cannot prove or disprove the property due to timeouts or exceptions.
- Inapplicable: The property does not apply to the project.

Detailed Results For Contract sLGNS (projects/ORIGIN/contracts/1/sERC20.sol) In Commit 0cb83d38b66ec4c1a7f5856b9b5d7a4c5ff283e3

Verification of ERC-20 Compliance

Detailed Results for Function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-succeed-always	● True	
erc20-totalsupply-change-state	● True	
erc20-totalsupply-correct-value	● True	

Detailed Results for Function `approve`

Property Name	Final Result	Remarks
erc20-approve-succeed-normal	● True	
erc20-approve-false	● True	
erc20-approve-correct-amount	● True	
erc20-approve-revert-zero	● False	
erc20-approve-never-return-false	● True	

Detailed Results for Function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-correct-value	● True	
erc20-balanceof-change-state	● True	
erc20-balanceof-succeed-always	● False	

Detailed Results for Function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-succeed-always	● True	
erc20-allowance-correct-value	● True	
erc20-allowance-change-state	● True	

Detailed Results for Function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-never-return-false	 True	
erc20-transferfrom-fail-exceed-balance	 False	
erc20-transferfrom-false	 True	
erc20-transferfrom-revert-zero-argument	 False	
erc20-transferfrom-fail-exceed-allowance	 True	
erc20-transferfrom-correct-allowance	 True	
erc20-transferfrom-correct-amount	 False	

Detailed Results for Function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-false	 True	
erc20-transfer-never-return-false	 True	
erc20-transfer-revert-zero	 False	
erc20-transfer-exceed-balance	 False	
erc20-transfer-correct-amount	 False	

**Detailed Results For Contract sLGNS (projects/ORIGIN/contracts/1/sERC20.sol) In Commit
028eab00efb2f9e510342c254006b2fbd0c778bc**

Verification of ERC-20 Compliance

Detailed Results for Function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-revert-zero-argument	False	
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-correct-allowance	True	
erc20-transferfrom-correct-amount	False	
erc20-transferfrom-false	True	
erc20-transferfrom-never-return-false	True	
erc20-transferfrom-fail-exceed-balance	False	

Detailed Results for Function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-exceed-balance	False	
erc20-transfer-correct-amount	False	
erc20-transfer-false	True	
erc20-transfer-revert-zero	False	
erc20-transfer-never-return-false	True	

Detailed Results for Function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-change-state	True	
erc20-balanceof-succeed-always	False	
erc20-balanceof-correct-value	True	

Detailed Results for Function `approve`

Property Name	Final Result	Remarks
erc20-approve-false	● True	
erc20-approve-correct-amount	● True	
erc20-approve-succeed-normal	● True	
erc20-approve-revert-zero	● False	
erc20-approve-never-return-false	● True	

Detailed Results for Function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-succeed-always	● True	
erc20-allowance-change-state	● True	
erc20-allowance-correct-value	● True	

Detailed Results for Function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-correct-value	● True	
erc20-totalsupply-succeed-always	● True	
erc20-totalsupply-change-state	● True	

APPENDIX | ORIGIN

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.
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 ERC-20 Properties

Properties related to function `totalSupply`

erc20-totalsupply-change-state

The `totalSupply` function in contract sLGNS must not change any state variables.

Specification:

```
assignable \nothing;
```

erc20-totalsupply-correct-value

The `totalSupply` function must return the value that is held in the corresponding state variable of contract sLGNS.

Specification:

```
ensures \result == totalSupply();
```

erc20-totalsupply-succeed-always

The function `totalSupply` must always succeeds, assuming that its execution does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `approve`

erc20-approve-correct-amount

All non-reverting calls of the form `approve(spender, amount)` that return `true` must correctly update the allowance mapping according to the address `msg.sender` and the values of `spender` and `amount`.

Specification:

```
requires spender != address(0);  
ensures \result ==> allowance(msg.sender, \old(spender)) == \old(amount);
```

erc20-approve-false

If function `approve` returns `false` to signal a failure, it must undo all state changes that it incurred before returning to the caller.

Specification:

```
ensures !\result ==> \assigned (\nothing);
```

erc20-approve-never-return-false

The function `approve` must never returns `false`.

Specification:

```
ensures \result;
```

erc20-approve-revert-zero

All calls of the form `approve(spender, amount)` must fail if the address in `spender` is the zero address.

Specification:

```
ensures \old(spender) == address(0) ==> !\result;
```

erc20-approve-succeed-normal

All calls of the form `approve(spender, amount)` must succeed, if

- the address in `spender` is not the zero address and
- the execution does not run out of gas.

Specification:

```
requires spender != address(0);
ensures \result;
reverts_only_when false;
```

Properties related to function `balanceOf`

erc20-balanceof-change-state

Function `balanceOf` must not change any of the contract's state variables.

Specification:

```
assignable \nothing;
```

erc20-balanceof-correct-value

Invocations of `balanceOf(owner)` must return the value that is held in the contract's balance mapping for address `owner`.

Specification:

```
ensures \result == balanceOf(\old(account));
```

erc20-balanceof-succeed-always

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

Specification:

```
reverts_only_when false;
```

Properties related to function `allowance`

erc20-allowance-change-state

Function `allowance` must not change any of the contract's state variables.

Specification:

```
assignable \nothing;
```

erc20-allowance-correct-value

Invocations of `allowance(owner, spender)` must return the allowance that address `spender` has over tokens held by address `owner`.

Specification:

```
ensures \result == allowance(\old(owner), \old(spender));
```

erc20-allowance-succeed-always

Function `allowance` must always succeed, assuming that its execution does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `transferFrom`

erc20-transferfrom-correct-allowance

All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` must decrease the allowance for address `msg.sender` over address `from` by the value in `amount`.

Specification:

```
ensures \result ==> allowance(\old(sender), msg.sender) == \old(allowance(sender,
msg.sender)) - \old(amount)
|| (allowance(\old(sender), msg.sender) == \old(allowance(sender,
msg.sender)) && \old(allowance(sender, msg.sender)) == type(uint256).max);
```

erc20-transferfrom-correct-amount

All invocations of `transferFrom(from, dest, amount)` that succeed and that return `true` subtract the value in `amount` from the balance of address `from` and add the same value to the balance of address `dest`.

Specification:

```
requires recipient != sender;
requires balanceOf(recipient) + amount <= type(uint256).max;
ensures \result ==> balanceOf(\old(recipient)) == \old(balanceOf(recipient) +
amount)
&& balanceOf(\old(sender)) == \old(balanceOf(sender) - amount);
also
requires recipient == sender;
ensures \result ==> balanceOf(\old(recipient)) == \old(balanceOf(recipient));
```

erc20-transferfrom-fail-exceed-allowance

Any call of the form `transferFrom(from, dest, amount)` with a value for `amount` that exceeds the allowance of address `msg.sender` must fail.

Specification:

```
requires msg.sender != sender;  
requires amount > allowance(sender, msg.sender);  
ensures !\result;
```

erc20-transferfrom-fail-exceed-balance

Any call of the form `transferFrom(from, dest, amount)` with a value for `amount` that exceeds the balance of address `from` must fail.

Specification:

```
requires amount > balanceOf(sender);  
ensures !\result;
```

erc20-transferfrom-false

If `transferFrom` returns `false` to signal a failure, it must undo all incurred state changes before returning to the caller.

Specification:

```
ensures !\result ==> \assigned (\nothing);
```

erc20-transferfrom-never-return-false

The `transferFrom` function must never return `false`.

Specification:

```
ensures \result;
```

erc20-transferfrom-revert-zero-argument

All calls of the form `transferFrom(from, dest, amount)` must fail for transfers from or to the zero address.

Specification:

```
ensures \old(sender) == address(0) ==> !\result;  
also  
ensures \old(recipient) == address(0) ==> !\result;
```

Properties related to function `transfer`

erc20-transfer-correct-amount

All non-reverting invocations of `transfer(recipient, amount)` that return `true` must subtract the value in `amount` from the balance of `msg.sender` and add the same value to the balance of the `recipient` address.

Specification:

```
requires recipient != msg.sender;
requires balanceOf(recipient) + amount <= type(uint256).max;
ensures \result ==> balanceOf(recipient) == \old(balanceOf(recipient) + amount)
&& balanceOf(msg.sender) == \old(balanceOf(msg.sender) - amount);
also
requires recipient == msg.sender;
ensures \result ==> balanceOf(msg.sender) == \old(balanceOf(msg.sender));
```

erc20-transfer-exceed-balance

Any transfer of an amount of tokens that exceeds the balance of `msg.sender` must fail.

Specification:

```
requires amount > balanceOf(msg.sender);
ensures !\result;
```

erc20-transfer-false

If the `transfer` function in contract `SLGNS` fails by returning `false`, it must undo all state changes it incurred before returning to the caller.

Specification:

```
ensures !\result ==> \assigned (\nothing);
```

erc20-transfer-never-return-false

The transfer function must never return `false` to signal a failure.

Specification:

```
ensures \result;
```

erc20-transfer-revert-zero

Any call of the form `transfer(recipient, amount)` must fail if the recipient address is the zero address.

Specification:

```
ensures \old(recipient) == address(0) ==> !\result;
```

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.

