Smart Contract
Security Audit Report

Prepared for Litra Finance

Prepared by Supremacy

March 23, 2025

Contents

1 Introduction

Given the opportunity to review the design document and related codebase of the Litra Finance, we outline in the report our systematic approach to evaluate potential security issues in the smart contract(s) implementation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Client

Litra Finance is an NFT liquidity protocol that wraps NFT into fungible ERC20 tokens to improve trading accuracy and provide more basic liquidity targets. Combining an automated centralized liquidity AMM with a customized curve provides traders with the ultimate trading experience with low slippage, while reducing impermanent losses for liquidity providers. The introduction of the veToken model encourages liquidity providers to actively and consistently provide liquidity for greater returns, which also aligns the interests of all parties.

Item	**Description**
Client	Litra Finance
Type	Smart Contract
Languages	Solidity
Platform	EVM-compatible

1.2 Audit Scope

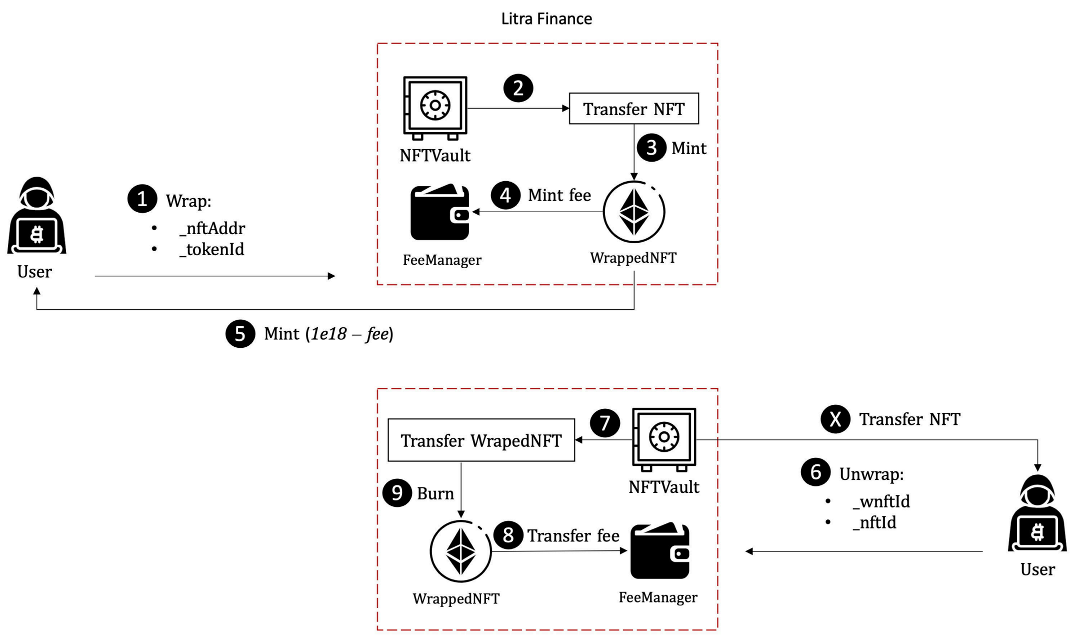
In the following, we show the given compressed file with codebase and its checksum value before and after auditing.

- litra-contracts.rar (SHA256: 0a3e607cfff4605d061e...fb4019b0dc4ce1b929d8)
- litra-contracts.rar (SHA256: 9303970128295bc8c223...1798eca9a38f97643b5b)

1.3 Changelogs

Version	**Date**	**Description**
0.1	April 07, 2023	Initial Draft
0.2	April 08, 2023	Release Candidate #1
1.0	April 23, 2023	Final Release
1.1	March 23, 2025	Post-Final Release #1

1.4 Threat Model



Litra Finance is an NFT liquidity protocol, and within the scope of observable security audits its main functions are the components NFTVault, FeeManager and Admin.

As shown above, this involves multiple interactions between a user who (wraps) his NFT into a WrappedNFT via Litra Finance and a user who (unwraps) his WrappedNFT into an NFT via Litra Finance. **During the audit, we assume the user could be malicious, which means all messages sent to Litra Finance are untrusted.**

We enumerated the attack surface based on this assumption.

1.5 About Us

Supremacy is a leading blockchain security firm, composed of industry hackers and academic researchers, provide top-notch security solutions through our technology precipitation and innovative research.

We are reachable at X (https://x.com/SupremacyHQ), or Email (contact@supremacy.email).

1.6 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- Likelihood represents the likelihood of a finding to be triggered or exploited in practice
- Impact specifies the technical and business-related consequences of a finding
- Severity is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Severity

Impact	*High*	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	*Medium*	*Low*
			Likelihood	

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

2 Findings

The table below summarizes the findings of the audit, including status and severity details.

ID	Severity	Description	Status
1	Critical	WrappedNFT forcible minting	Fixed
2	Critical	Theft of any user’s NFT	Fixed
3	Medium	Potential fee loss	Fixed
4	Medium	Centralization risk	Confirmed
5	Low	Unchecked zero-address	Fixed
6	Low	Unchecked return values	Fixed
7	Informational	Missing event records	Fixed
8	Informational	Best Practices	Fixed
9	Informational	Best Practices	Fixed

2.1 Critical

1. WrappedNFT forcible minting [Medium]

```
Severity: Critical                    Likelihood: High                  Impact: High
Status: Fixed
```

Description

`NFTVault::wrap()` is a function that converts ERC721 (NFT) to ERC20 (WrappedNFT). However, since `#L90` assigns wnftId to `wnftIds[WrappedNFT]` and `_nftAddr` is controllable, resulting in WrappedNFT being arbitrarily mint.

1. The hacker selects an already existing WrappedNFT series and calls `NFTVault::wrap(WrappedNFT, 0)`.
2. Then `#L56` of `wrap()` will call `WrappedNFT::transferFrom()` externally, and since WrappedNFT is a standard ERC20 Token, it can be executed normally, but without actually transferring the Token, because `_value` is `0`.
3. Since WrappedNFT can obtain `wnftId` through `#L61`, it will not enter the `CREATE` procedure, but the `else` condition, and the `wnft` obtained through `#L94` is WrappedNFT itself, so the subsequent procedure, will be directly for the `caller` Mint WrappedNFT.

```
47      /**
48          @notice Wrap a NFT(IERC721) into a ERC20 token.
49          @param _nftAddr address of NFT contract
50          @param _tokenId token id of the NFT
51       */
52      function wrap (
53          address _nftAddr,
54          uint256 _tokenId
55      ) external payable nonReentrant {
56          IERC721(_nftAddr).transferFrom(msg.sender, address(this), _tokenId);
57          // Save nft record
58          uint256 recordId = wrappedNfts.length;
59          wrappedNfts.push(WrappedNFTInfo(_nftAddr, _tokenId, true));
60          // Get FT Info
61          uint256 wnftId = wnftIds[_nftAddr];
62          address wnft;
63          if(wnftId == 0) {
64              // Create a new FT
65              string memory wnftName;
66              string memory wnftSymbol;
67              (bool succeed, bytes memory result) =
    _nftAddr.call(abi.encodeWithSignature("name()"));
68              if(succeed) {
69                  string memory nftName = abi.decode(result, (string));
70                  wnftName = string(abi.encodePacked(nftName, " Wrapped NFT"));
71              } else {
72                  wnftName = string(abi.encodePacked("Litra FT#", wnftId));
73              }
74              (succeed, result) =
    _nftAddr.call(abi.encodeWithSignature("symbol()"));
75              if(succeed) {
76                  string memory nftSymbol = abi.decode(result, (string));
77                  wnftSymbol = string(abi.encodePacked(nftSymbol, "wnft"));
78              } else {
79                  wnftSymbol = string(abi.encodePacked("LWNFT#", wnftId));
80              }
81              wnft = address(new WrappedNFT(wnftName, wnftSymbol));
82              // get ftId
83              uint256 _nextWnftId = nextWnftId;
84              wnftId = _nextWnftId;
85              _nextWnftId ++;
86              nextWnftId = _nextWnftId;
87              // storage
88              wnfts[wnftId] = WNFTInfo(_nftAddr, wnft);
89              wnftIds[_nftAddr] = wnftId;
90              wnftIds[wnft] = wnftId;
91
92              emit CreateWrappedNFT(_nftAddr, wnftId, wnft);
93          } else {
94              wnft = wnfts[wnftId].wnftAddr;
95          }
96          // bound FT and NFT
97          _nfts[wnftId].add(recordId);
98          // mint and charge fee
99          uint256 fee;
100         if(address(feeManager) != address(0)) {
101             fee = feeManager.wrapFee(wnft);
102         }
103         if(fee > 0) {
104             WrappedNFT(wnft).mint(address(feeManager), fee);
105         }
106         WrappedNFT(wnft).mint(msg.sender, 1e18 - fee);
107
108         emit Wrap(msg.sender, wnftId, recordId);
109     }
```

NFTVault.sol

Recommendation

Delete `#L90` from `NFTVault.sol`.

2. Theft of any user's NFT [Critical]

Severity: Critical   Likelihood: High   Impact: High

Status: Fixed

Description

Based on the premise of **Critical-1**, a hacker can force the minting of any number of WrappedNFTs under a certain NFT series. However, in the `NFTVault::unwrap()` function, it allows the user to submit `1e18` WrappedNFTs and redeem the NFTs of that series. thus, the hacker can premeditatedly obtain all the deposited NFTVault in NFTVault and thus theft all users' NFTs by calling `NFTVault::unwrap()`.

```
    /**
        @notice Redeem nft from vault and burn one FT
        @param _wnftId index of fts
        @param _nftId Greate than or equal 0 to redeem a designated nft with a more fees
                        Less than 0 to redeem a recent fungiblized nft with a normal fee
     */
    function unwrap(uint256 _wnftId, uint256 _nftId) external payable nonReentrant {
        WNFTInfo memory ftInfo = wnfts[_wnftId];
        require(ftInfo.nftAddr != address(0), "Invalid FT");
        require(WrappedNFT(ftInfo.wnftAddr).balanceOf(msg.sender) >= 1e18, "Insufficient ft");
        require(_nfts[_wnftId].length() > 0, "No NFT in vault");
        require(_nfts[_wnftId].contains(uint256(_nftId)), "Invalid nftId");
        // burn ft and charge fee
        uint256 fee;
        if(address(feeManager) != address(0)) {
            fee = feeManager.unwrapFee(ftInfo.wnftAddr);
        }
        if(fee > 0) {
            WrappedNFT(ftInfo.wnftAddr).transferFrom(msg.sender, address(feeManager), fee);
        }
        WrappedNFT(ftInfo.wnftAddr).transferFrom(msg.sender, address(this), 1e18);
        WrappedNFT(ftInfo.wnftAddr).burn(1e18);
        // return nft
        WrappedNFTInfo memory nftInfo = wrappedNfts[_nftId];
        wrappedNfts[_nftId].inVault = false;
        _nfts[_wnftId].remove(_nftId);
        IERC721(nftInfo.nftAddr).safeTransferFrom(address(this), msg.sender, nftInfo.tokenId);

        emit Unwrap(msg.sender, _wnftId, _nftId);
    }
```

NFTVault.sol

Recommendation

Refer to the **Critical-1** recommendation.

2.2 Medium

3. Potential fee loss [Medium]

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

`Status: Fixed`

Description

It is not recommended to leave the initial fee to the user to set, because due to the atomic nature of the transaction, it is not possible to charge a fee for WrappedNFTs created within a single transaction anyway. If the user sets both `Wrap` and `Unwrap` fees to `0` after the WrappedNFT is created, no fee will be charged during the window until the parameterAdmin sets the fee again, thus, causing some financial loss.

```
    /**
        @notice Set fee for wrapping.
        Anyone can make the first setting,but generally the first maker will be creator of wnft.
        After first setting, only parameter admin can change
```

```
     */
    function setWrapFee(address _wnft, uint256 _fee) external {
        Fee memory fee = _wrapFee[_wnft];
        require(!fee.initialized || msg.sender == parameterAdmin, "! parameter
admin");
        _wrapFee[_wnft] = Fee(true, _fee);
    }

    /**
        @notice Set fee for unwrapping.
        Anyone can make the first setting,but generally the first maker will be
creator of wnft.
        After first setting, only parameter admin can change
     */
    function setUnwrapFee(address _wnft, uint256 _fee) external {
        Fee memory fee = _unwrapFee[_wnft];
        require(!fee.initialized || msg.sender == parameterAdmin, "! parameter
admin");
        _unwrapFee[_wnft] = Fee(true, _fee);
    }
```

FeeManager.sol

Recommendation

When the user calls `NFTVault::Wrap()` or `NFTVault::Unwrap()`, call `FeeManager::setWrapFee()` externally to set the fee instantly before NFTVault transfers the fee (need to add access control in FeeManager).

4. Centralization risk [Medium]

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

```
Status: Confirmed
```

Description

In the Litra Finance, privileged accounts exist that play a key role in managing and regulating the operation of the entire system (e.g., configuring various parameters and setting stopped parameters). It also has the privilege of controlling or managing the flow of assets managed by the protocol.

Our analysis shows that privileged accounts need to be scrutinized. In the following, we will examine privileged accounts and the associated privileged access in the current contract.

Note that if the privileged owner account is a plain EOA, this may be worrisome and pose counter-party risk to the protocol users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

```
contract OwnershipAdminManaged {
    address public ownershipAdmin;
    address public futureOwnershipAdmin;

    constructor(address _o) {
        ownershipAdmin = _o;
    }

    modifier onlyOwnershipAdmin {
        require(msg.sender == ownershipAdmin, "! ownership admin");
        _;
    }

    function commitOwnershipAdmin(address _o) external onlyOwnershipAdmin {
        futureOwnershipAdmin = _o;
    }

    function applyOwnershipAdmin() external {
        require(msg.sender == futureOwnershipAdmin, "Access denied!");
        ownershipAdmin = futureOwnershipAdmin;
    }
}
```

OwnershipAdminManaged.sol

```
pragma solidity ^0.8.0;

import "./OwnershipAdminManaged.sol";

abstract contract EmergencyAdminManaged is OwnershipAdminManaged {
    address public emergencyAdmin;
    address public futureEmergencyAdmin;

    constructor(address _e) {
        emergencyAdmin = _e;
    }

    modifier onlyEmergencyAdmin {
        require(msg.sender == emergencyAdmin, "! emergency admin");
        _;
    }

    function commitEmergencyAdmin(address _e) external onlyEmergencyAdmin {
        futureEmergencyAdmin = _e;
    }

    function applyEmergencyAdmin() external {
        require(msg.sender == futureEmergencyAdmin, "! emergency admin");
        emergencyAdmin = futureEmergencyAdmin;
    }
}
```

EmergencyAdminManaged.sol

```
pragma solidity ^0.8.0;

import "./OwnershipAdminManaged.sol";
```

```

abstract contract ParameterAdminManaged is OwnershipAdminManaged {
    address public parameterAdmin;
    address public futureParameterAdmin;

    constructor(address _e) {
        parameterAdmin = _e;
    }

    modifier onlyParameterAdmin {
        require(msg.sender == parameterAdmin, "! parameter admin");
        _;
    }

    function commitParameterAdmin(address _p) external onlyOwnershipAdmin {
        futureParameterAdmin = _p;
    }

    function applyParameterAdmin() external {
        require(msg.sender == futureParameterAdmin, "Access denied!");
        parameterAdmin = futureParameterAdmin;
    }
}
```

ParameterAdminManaged.sol

Recommendation

Initially onboarding could can use multisign wallets or timelocks to initially mitigate centralization risks, but as a long-running protocol, we recommend eventually transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks.

Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

2.3 Low

5. Unchecked zero-address [Low]

`Severity: Low` `Likelihood: Low` `Impact: Low`

`Status: Fixed`

Description

Adding the requirement for non-zero address to target.

```
EmergencyAdminManaged::commitEmergencyAdmin()
OwnershipAdminManaged::commitOwnershipAdmin()
ParameterAdminManaged::commitParameterAdmin()
```

Recommendation

Consider adding zero address validation and non-previous address validation.

6. Unchecked return values [Low]

Severity: Low   Likelihood: Low   Impact: Low

Status: Undetermined

Description

The NFTVault contract uses the `EnumerableSet` library to `add` and `remove` values to the set. Both functions return boolean values when they're called.

```
    /**
        @notice Wrap a NFT(IERC721) into a ERC20 token.
        @param _nftAddr address of NFT contract
        @param _tokenId token id of the NFT
     */
    function wrap (
        address _nftAddr,
        uint256 _tokenId
    ) external payable nonReentrant {
        IERC721(_nftAddr).transferFrom(msg.sender, address(this), _tokenId);
        // Save nft record
        uint256 recordId = wrappedNfts.length;
        wrappedNfts.push(WrappedNFTInfo(_nftAddr, _tokenId, true));
        // Get FT Info
        uint256 wnftId = wnftIds[_nftAddr];
        address wnft;
        if(wnftId == 0) {
            // Create a new FT
            string memory wnftName;
            string memory wnftSymbol;
            (bool succeed, bytes memory result) = _nftAddr.call(abi.encodeWithSignature("name()"));
            if(succeed) {
                string memory nftName = abi.decode(result, (string));
                wnftName = string(abi.encodePacked(nftName, " Wrapped NFT"));
            } else {
                wnftName = string(abi.encodePacked("Litra FT#", wnftId));
            }
            (succeed, result) = _nftAddr.call(abi.encodeWithSignature("symbol()"));
            if(succeed) {
                string memory nftSymbol = abi.decode(result, (string));
                wnftSymbol = string(abi.encodePacked(nftSymbol, "wnft"));
            } else {
                wnftSymbol = string(abi.encodePacked("LWNFT#", wnftId));
            }
            wnft = address(new WrappedNFT(wnftName, wnftSymbol));
            // get ftId
            uint256 _nextWnftId = nextWnftId;
            wnftId = _nextWnftId;
            _nextWnftId ++;
            nextWnftId = _nextWnftId;
            // storage
            wnfts[wnftId] = WNFTInfo(_nftAddr, wnft);
            wnftIds[_nftAddr] = wnftId;
            wnftIds[wnft] = wnftId;

            emit CreateWrappedNFT(_nftAddr, wnftId, wnft);
        } else {
            wnft = wnfts[wnftId].wnftAddr;
        }
        // bound FT and NFT
        _nfts[wnftId].add(recordId);
        // mint and charge fee
        uint256 fee;
        if(address(feeManager) != address(0)) {
            fee = feeManager.wrapFee(wnft);
        }
        if(fee > 0) {
            WrappedNFT(wnft).mint(address(feeManager), fee);
        }
        WrappedNFT(wnft).mint(msg.sender, 1e18 - fee);

        emit Wrap(msg.sender, wnftId, recordId);
    }
```

NFTVault.sol

```
    /**
        @notice Redeem nft from vault and burn one FT
        @param _wnftId index of fts
        @param _nftId Greate than or equal 0 to redeem a designated nft with a
more fees
                            Less than 0 to redeem a recent fungiblized nft with a
normal fee
     */
    function unwrap(uint256 _wnftId, uint256 _nftId) external payable
nonReentrant {
        WNFTInfo memory ftInfo = wnfts[_wnftId];
        require(ftInfo.nftAddr != address(0), "Invalid FT");
        require(WrappedNFT(ftInfo.wnftAddr).balanceOf(msg.sender) >= 1e18,
"Insufficient ft");
        require(_nfts[_wnftId].length() > 0, "No NFT in vault");
        require(_nfts[_wnftId].contains(uint256(_nftId)), "Invalid nftId");
        // burn ft and charge fee
        uint256 fee;
        if(address(feeManager) != address(0)) {
            fee = feeManager.unwrapFee(ftInfo.wnftAddr);
        }
        if(fee > 0) {
            WrappedNFT(ftInfo.wnftAddr).transferFrom(msg.sender,
address(feeManager), fee);
        }
        WrappedNFT(ftInfo.wnftAddr).transferFrom(msg.sender, address(this),
1e18);
        WrappedNFT(ftInfo.wnftAddr).burn(1e18);
        // return nft
        WrappedNFTInfo memory nftInfo = wrappedNfts[_nftId];
        wrappedNfts[_nftId].inVault = false;
        _nfts[_wnftId].remove(_nftId);
        IERC721(nftInfo.nftAddr).safeTransferFrom(address(this), msg.sender,
nftInfo.tokenId);

        emit Unwrap(msg.sender, _wnftId, _nftId);
    }
```

Therefore, there should be a check to validate that the addition or removal from the set was correct. Otherwise, every time the function is called, the owner will have to check using getter functions.

NFTVault.sol

Recommendation

Revise the code logic accordingly.

2.4 Informational

7. Missing event records [Informational]

`Status: Fixed`

Description

In the `EmergencyAdminManaged`, `OwnershipAdminManaged`, `ParameterAdminManaged`, and `Stoppable` contracts, privileged accounts can set Admin privileges and `stopped` status through privileged functions respectively, but no event logging is performed. And in `FeeManager` contract, user will call `setWrapFee()` & `setUnwrapFee()` to change the status, but no event logging.

```
pragma solidity ^0.8.0;

import "./OwnershipAdminManaged.sol";

abstract contract EmergencyAdminManaged is OwnershipAdminManaged {
    address public emergencyAdmin;
    address public futureEmergencyAdmin;

    constructor(address _e) {
        emergencyAdmin = _e;
    }

    modifier onlyEmergencyAdmin {
        require(msg.sender == emergencyAdmin, "! emergency admin");
        _;
    }

    function commitEmergencyAdmin(address _e) external onlyEmergencyAdmin {
        futureEmergencyAdmin = _e;
    }

    function applyEmergencyAdmin() external {
        require(msg.sender == futureEmergencyAdmin, "! emergency admin");
        emergencyAdmin = futureEmergencyAdmin;
    }
}

```

EmergencyAdminManaged.sol

Events are important because off-chain monitoring tools rely on them to index important state changes to the smart contract(s).

Recommendation

Consider emitting events when state changes are performed in the `EmergencyAdminManaged`, `OwnershipAdminManaged`, `ParameterAdminManaged`, `Stoppable` and `FeeManager` contract.

8. Best Practices [Informational]

`Status: Fixed`

Description

Some storage variables should be immutable Marking these as immutable (as they never change outside the constructor) would avoid them taking space in the storage.

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

import "./admin/Stoppable.sol";
import "../interfaces/IBurner.sol";
import "../interfaces/IFeeManager.sol";
import "./admin/ParameterAdminManaged.sol";

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract FeeManager is IFeeManager, Stoppable, ParameterAdminManaged {
    struct Fee {
        bool initialized;
        uint256 value;
    }

    address public vault;
    ...

```

FeeManager.sol

Recommendation

Revise the code logic accordingly.

9. Best Practices [Informational]

`Status: Fixed`

Description

In the `EmergencyAdminManaged`, `OwnershipAdminManaged` and `ParameterAdminManaged` contracts, privileged accounts can each set administrative privileges via privileged functions, where a transition account `future` exists so that the `future` account in the future can be upgraded to administrator by calling the `applyAdmin()` privilege function.

However, the call to `applyAdmin()` does not reset future, making the calling account both `Admin` and `future`.

```
pragma solidity ^0.8.0;

import "./OwnershipAdminManaged.sol";

abstract contract EmergencyAdminManaged is OwnershipAdminManaged {
    address public emergencyAdmin;
    address public futureEmergencyAdmin;

    constructor(address _e) {
        emergencyAdmin = _e;
    }

    modifier onlyEmergencyAdmin {
        require(msg.sender == emergencyAdmin, "! emergency admin");
        _;
    }

    function commitEmergencyAdmin(address _e) external onlyEmergencyAdmin {
        futureEmergencyAdmin = _e;
    }

    function applyEmergencyAdmin() external {
        require(msg.sender == futureEmergencyAdmin, "! emergency admin");
        emergencyAdmin = futureEmergencyAdmin;
    }
}
```

EmergencyAdminManaged.sol

Recommendation

Revise the code logic accordingly.

3 Disclaimer

This security audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. This security audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues, also cannot make guarantees about any additional code added to the assessed project after the audit version. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contract(s). Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.