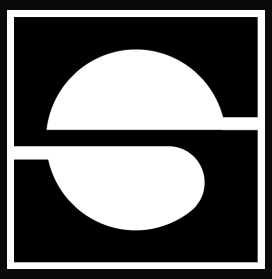


Security Assessment

Final Report



Saturn-sUSDat v2

August 2026

Prepared for Saturn Organization

certora

Saturn

Table of Contents

Project Summary **3**
- Project Scope 3
- Project Overview 3
- Protocol Overview 4

Assessment Methodology **6**

Threat and Security Overview **7**
- Findings Summary 9
- Severity Matrix 9

Detailed Findings **10**
- Low Severity Issues 12
  - L-01: totalAssets reverts on oracle failure, violating EIP-4626 12
  - L-02: InProgress withdrawal requests are permanently stuck after the migration 13
  - L-03: processRequests reverts on any non-Requested entry, letting any user permanently grief redemption settlement via cancelRequest 14
  - L-04: ERC-4626 preview and max functions do not reflect the redemption fee or the disabled synchronous exit, overstating sUSDat exit value to integrators 15
  - L-05: ERC-165 omits IWithdrawalQueueERC721 17
  - L-06: Vested surplus inflates NAV but is siloed from the redemption liquidity check, causing false InsufficientLiquidity 18
  - L-07: Redemption-fee accretion raises PPS per settlement; request splitting and operator ordering decide who captures it 20
  - L-08: Just-in-time deposits can capture redemption fees from queued holders 22
  - L-09: setMaxRewardsBps has no upper bound, allowing unbounded reward injection 22
  - L-10: Blacklisted approved operators retain authority over other users' tokens and withdrawal claims. 23
  - L-11: STRConModule.setOracle performs no live price smoke-test, so a rotation can immediately brick the whole vault 24
- Informational Severity Issues 26
  - I-01: setMaxApiStaleness / setDeviationBps accept 0, permanently bricking `getPrice()` 26

Disclaimer **27**

About Certora **27**

certora

Saturn

Project Summary

Project Scope

Project Name	Repository Link	Commit Hash	Platform
Saturn-sUSDat v2	https://github.com/saturn-organization/saturn-yield-dollar	initial commit 4157ff8 final commit f9bb4f9	Solidity

Project Overview

This document describes the security review of **Saturn-sUSDat v2**. The work was undertaken from **July 27th, 2026** to **August 5th, 2026**.

The following contract list is included in our scope:

src/v2/StakedUSDat.sol
src/v2/WithdrawalQueueERC721.sol
src/v2/STRConExecutionPolicy.sol
src/v2/libraries/STRConTradeExecutionLogic.sol
src/v2/modules/STRCon/STRConModule.sol
src/v2/modules/STRCon/STRConPriceOracle.sol
src/v2/modules/MirrorSTRC/STRCMirrorModule.sol
src/v2/interfaces/IStakedUSDat.sol
src/v2/interfaces/IWithdrawalQueueERC721.sol
src/v2/interfaces/ISTRConExecutionPolicy.sol
src/v2/interfaces/IUSDat.sol
src/v2/interfaces/IFreezable.sol
src/v2/interfaces/ISwapFacility.sol
src/v2/interfaces/IERC20PermitExtended.sol
src/v2/interfaces/modules/IAccountingModule.sol

src/v2/interfaces/modules/ITradableModule.sol
src/v2/interfaces/modules/ISTRConModule.sol
src/v2/interfaces/modules/ISTRCMirrorModule.sol
src/v2/interfaces/oracles/ISTRConPriceOracle.sol
src/v2/interfaces/oracles/IStrcPriceOracle.sol
src/v2/interfaces/oracles/IPriceOracle.sol
src/v2/interfaces/oracles/ISyntheticSharesOracle.sol

The team performed a manual audit of all the files in scope**.** During the manual audit, the Certora team discovered bugs in the code, as listed on the following page.

Protocol Overview

Saturn is a DeFi protocol that issues a USD-denominated stablecoin (USDat) and a yield-bearing vault token (sUSDat). USDat is built on M0's M token infrastructure as a JMIExtension, minted 1:1 against stable assets by KYC-verified users.

Users can permissionlessly deposit USDat into the sUSDat vault (an ERC-4626 contract) to gain synthetic exposure to STRC, MicroStrategy's NASDAQ-listed preferred equity, which trades at approximately $100 and pays ~11% monthly cash dividends. v2 restructures how that exposure is held. Where v1 tracked STRC as a virtual balance reconciled off-chain, v2 moves the position on-chain to STRCon (Ondo's tokenized STRC) and splits the vault's backing into two fixed accounting modules. The STRCMirrorModule is a transitional, tokenless bridge that preserves the legacy STRC position - its balance is processor-attested and priced by the v1 oracle - and is retired once the position is migrated. The STRConModule is the sole token-backed, tradable module, it holds STRCon and is priced by a new two-feed oracle (STRConPriceOracle) that combines a Chainlink-compatible mark, an Ondo reference feed, and the Ondo sValue.

A trusted operator manages the vault's exposure through rotations: it buys and sells STRCon against the vault's USDat buffer through an off-chain execution vehicle (buy/sell), with each leg validated against the oracle price and rate-limited by a fixed execution policy - a per-trade tolerance and a continuously-refilling capacity bucket. STRC dividends and other yield are added through transferInSurplus and transferInRewards and vest linearly, so no single depositor can capture an in-flight tranche. The vault's totalAssets() is denominated in USD and reflects the on-chain USDat buffer, vested surplus, and the recognized USD value of the two modules, pricing

is fail-closed, so any module that cannot be priced reverts totalAssets() and halts value-sensitive operations.

Withdrawals from sUSDat are asynchronous and handled through a dedicated WithdrawalQueue implemented as an ERC-721 contract; synchronous withdraw and redeem are disabled. Users submit redemption requests that escrow shares into the queue and specify a minimum net share price (minSharePrice). The operator processes and settles requests from the USDat buffer, after which users claim their USDat. Unlike v1, request owners may cancel an open request, and compliance can blacklist, honor USDat freezes, and seize positions. A one-time, timelocked migration retires the mirror and recognizes the migrated STRCon, after which the vault is backed by STRCon alone.

Assessment Methodology

Our assessment approach combines design level analysis with a deep review of the implementation to ensure that a protocol is secure, economically sound, and behaves as intended under realistic conditions.

At the design level, we evaluate the architecture, the economic assumptions behind the protocol, and the safety properties that should hold independently of a specific chain or environment. This process includes reviewing internal and cross protocol interactions, state transition flows, trust boundaries, and any mechanism that could be exploited to extract value, deny service, or alter core system behavior. At this stage, a focused threat modelling exercise helps identify key attack surfaces and adversarial capabilities relevant to the system. Design level issues often relate to incentive structures, governance implications, or systemic behavior that emerges under adversarial conditions.

Implementation analysis focuses on the concrete behavior of the code within the execution model of the target chain. This involves reviewing the correctness of logic, access control, state handling, arithmetic behavior, and the nuanced behaviors of the chain environment. Familiar classes of vulnerabilities such as reentrancy conditions, faulty permission checks, precision issues, or unsafe assumptions often surface at this layer. These findings require context aware reasoning that takes into account both the code and the architectural intent.

To support this analysis, the codebase is examined through repeated manual passes and supplemented by automated tools when appropriate. High-risk logic areas receive deeper scrutiny, invariants are validated against both design intent and actual implementation, and potential vulnerability leads are thoroughly investigated. Automated techniques such as static analysis, fuzzing, or symbolic execution may be used to complement manual review and provide additional insight.

Collaboration with the development team plays an important role throughout the audit. This helps confirm expected behaviors, clarify design assumptions, and ensure an accurate understanding of the protocol's intended operation. All findings are documented with clear reasoning, reproducible examples, and actionable recommendations. A follow up review is conducted to validate the applied fixes and verify that no regressions or secondary issues have been introduced.

Threat and Security Overview

The primary security invariant is that totalAssets() = usdatBalance + vestedSurplus + STRCMirrorModule.recognizedValue() + STRConModule.recognizedValue() must accurately reflect the protocol's backing at all times, and that redemptions burn shares only in exchange for their true NAV, funded from the vault's USDat buffer. Because backing is sourced through off-chain rotations and settlement is asynchronous, correctness relies on enforcing these invariants through on-chain validation - exact custody deltas, custody floors, and oracle-bounded tolerances, rather than atomic execution.

The most significant trust boundary is the value-path role set: the OPERATOR (rotations, reward and surplus injection, queue processing) and the PARAMETER_MANAGER (oracle, execution limits, fees, reward and vesting parameters, migration tolerance). Neither is behind the 5-day timelock that guards the DEFAULT_ADMIN (upgrades and migration), so both act with immediate effect. The protocol constrains them with a hard per-trade execution tolerance (≤5%), a refilling capacity bucket, custody floors, and fail-closed pricing, but between them these roles can still repoint the NAV oracle, inject attested value, and move the vault's assets, so they remain capable of influencing economic outcomes. The execution vehicle that fills rotations is team-owned and a separate address from the operator but in the same trust domain. Additional trusted roles include the Admin (pause, upgrades, migration - timelocked), the MARKET_MODE_MANAGER (market mode and elevated deposit fee), and Compliance (blacklist, freeze, seize).

The audit considered attack vectors including:

- Decimal mismatches between USDat (6 decimals), vault shares (18 decimals), STRCon, and oracle prices (8 decimals)
- Oracle failure scenarios - staleness during NASDAQ weekend closures, feed deviation, price-bound breaches, the Ondo sValue and pause dependency, and totalAssets() reverting
- Rotation value leakage within the execution tolerance and its accumulation over time
- Uncapped or manipulable reward injection into the mirror module (attested NAV with no custody floor)
- ERC-4626 inflation and donation attacks (mitigated by _decimalsOffset = 12 and exact-delta accounting)

- Withdrawal-queue state-machine edge cases - stuck legacy InProgress requests, the minUsdatReceived -> minSharePrice slot reinterpretation, batch-processing griefing, and NFT handling
- Compliance bypass via USDat-freeze evasion and blacklisted spender allowances
- The one-time STRC -> STRCon migration and its preconditions

All core v2 contracts were manually reviewed, including StakedUSDat, WithdrawalQueueERC721, STRConModule, STRConPriceOracle, STRCMirrorModule, STRConExecutionPolicy, and STRConTradeExecutionLogic. Invariants around share supply, NAV computation, rotation settlement, and withdrawal processing were analyzed across contract boundaries rather than in isolation.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	-	-	-
Low	11	11	9
Informational	1	1	1
Total	12	12	10

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
L-01	totalAssets() reverts on oracle failure, violating EIP-4626	Low	Acknowledged
L-02	InProgress withdrawal requests are permanently stuck after the migration	Low	Fixed
L-03	processRequests reverts on any non-Requested entry, letting any user permanently grief redemption settlement via cancelRequest	Low	Fixed
L-04	ERC-4626 preview and max functions do not reflect the redemption fee or the disabled synchronous exit, overstating sUSDat exit value to integrators	Low	Fixed
L-05	ERC-165 omits IWithdrawalQueueERC721	Low	Fixed
L-06	Vested surplus inflates NAV but is siloed from the redemption liquidity check, causing false InsufficientLiquidity	Low	Fixed
L-07	Redemption-fee accretion raises PPS per settlement; request splitting and operator ordering decide who captures it	Low	Fixed
L-08	Just-in-time deposits can capture redemption fees from queued holders	Low	Acknowledged

L-09	setMaxRewardsBps has no upper bound, allowing unbounded reward injection	Low	Fixed
L-10	Blacklisted approved operators retain authority over other users' tokens and withdrawal claims	Low	Fixed
L-11	STRConModule.setOracle performs no live price smoke-test, so a rotation can immediately brick the whole vault	Low	Fixed
I-01	setMaxApiStaleness / setDeviationBps accept 0, permanently bricking `getPrice()`	Informational	Fixed

Low Severity Issues

L-01: totalAssets reverts on oracle failure, violating EIP-4626

Severity: Low	Impact: Medium	Likelihood: Low
Files: StakedUSDat.sol	Status: Acknowledged	

Description:

EIP-4626 requires totalAssets() to never revert. Still, here it sums each module's recognizedValue(), which reverts whenever a module cannot be priced (oracle staleness, feed deviation, Ondo paused, sValue == 0, or price out of bounds). So totalAssets() and every view built on it (convertToAssets, previewDeposit/previewRedeem, share-price reads) revert on any pricing failure. This is a deliberate fail-closed design, and maxDeposit/maxMint catch it to return 0 for ERC-4626, but the core totalAssets()/preview surface still reverts, which breaks external integrators reading those views. If the sole post-migration STRCon oracle feed is ever removed/deprecated, deposits, redemptions, and rotations all halt until the oracle is repointed.

Recommendations:

Accepted fail-closed tradeoff, but document the EIP-4626 deviation for integrators, and confirm the recovery path - PARAMETER_MANAGER can setOracle to a working replacement and that a replacement price source will be available, so a permanently dead feed cannot brick the vault indefinitely.

Customer Response:

Acknowledged.

L-02: InProgress withdrawal requests are permanently stuck after the migration

Severity: Low	Impact: Low	Likelihood: Low
Files:	Status: Fixed	

Description:

v1 persists an InProgress request state. v2 removed that flow and has no function that accepts InProgress So any request that is InProgress at the moment of upgrade is permanently locked: it cannot be processed, claimed, or cancelled, and the user's escrowed shares are frozen. The spec acknowledges this ("locked forever") and relies on an off-chain checklist to convert such requests to Requested first, but nothing on-chain prevents the migration from running while InProgress requests exist.

Recommendations:

Handle InProgress the same as Requested in the queue functions, or add an owner/admin-callable InProgress to Requested converter that stays callable after migration so a forgotten request can always be rescued.

Customer Response:

Fixed in aad7908.

Fix Review:

Fix confirmed.

L-03: processRequests reverts on any non-Requested entry, letting any user permanently grief redemption settlement via cancelRequest

Severity: Low	Impact: Low	Likelihood: High
Files: src/v2/WithdrawalQueueERC721.sol	Status: Fixed	

Description:

processRequests hard-reverts the whole batch as soon as it encounters a request whose status is not Requested:

```
Request storage req = requests[tokenId];
require(req.status == RequestStatus.Requested, RequestNotOpen());
```

At the same time, cancelRequest is a fully permissionless action available to the current NFT owner of any open request, and it sets req.status = RequestStatus.Cancelled. Because the operator has to publish the list of tokenIds in a public transaction, any holder of a queued request that is included in the batch can watch the mempool and front-run the operator with cancelRequest(tokenId). The operator's transaction then reverts with RequestNotOpen() and no request in the batch settles — including all the requests that would otherwise have been fillable.The attack is extremely cheap and repeatable:1. The griefer only needs one request of MIN_REQUEST_SHARES (10 sUSDat ≈ 10 USDat of value, which is fully returned by the cancel),2. They can immediately create a new request (requestRedeem) and cancel it again against the next batch,cancellation charges no fee, so the only cost is gas.3. The result is an indefinite denial of service on the only exit path of the vault (withdraw/redeem are disabled), affecting every other user in the queue.

Recommendations:

Instead of reverting the entire transaction when a request is no longer in the Requested state, the processRequests function should gracefully skip the invalid request and continue processing the rest of the batch.

Customer Response:

Fixed in b9c290b.

Fix Review:

Fix confirmed.

L-04: ERC-4626 preview and max functions do not reflect the redemption fee or the disabled synchronous exit, overstating sUSDat exit value to integrators

Severity: Low	Impact: Low	Likelihood: High
Files: src/v2/StakedUSDat.sol	Status: Fixed	

Description:

StakedUSDat presents itself as an ERC-4626 vault (IStakedUSDat is IERC4626) and overrides withdraw, redeem, maxWithdraw, maxRedeem, previewDeposit and previewMint — but leaves previewWithdraw and previewRedeem at the inherited implementations, which are pure _convertToAssets / _convertToShares conversions with no fee applied:

```
/// @dev Disabled - use requestRedeem instead.
function redeem(uint256, address, address) public pure
override(ERC4626Upgradeable, IERC4626) returns (uint256) {
    revert OperationNotAllowed();
}
...
/// @dev Returns 0 when paused per ERC4626 spec.
function maxRedeem(address owner) public view override(ERC4626Upgradeable,
IERC4626) returns (uint256) {
    return paused() ? 0 : balanceOf(owner);
}
```

Three concrete deviations follow:

1. `previewRedeem` omits the redemption fee**.** The only exit path is `requestRedeem` → `redeemQueuedShares`, which deducts `redemptionFeeBps()` (`baseRedemptionFeeBps` or `elevatedRedemptionFeeBps`, up to `MAX_REDEMPTION_FEE_BPS = 500`) from the gross conversion before paying out. `previewRedeem(shares)` returns the *gross* amount and therefore overstates the realizable exit value by up to 5%. ERC-4626 explicitly requires preview functions to include all fees and slippage.
2. `maxRedeem` returns a non-zero value for a function that always reverts**.** ERC-4626 defines `maxRedeem` as the maximum share amount for which a call to `redeem` would not revert. Here `redeem` is `pure` and reverts unconditionally, so the only spec-correct value is `0`. Any generic ERC-4626 router or aggregator that does `redeem(maxRedeem(user), ...)` will revert.
3. `maxWithdraw` and `previewWithdraw` are mutually inconsistent. `maxWithdraw` is hard-coded to `0`, while `previewWithdraw(assets)` still returns a non-zero share count for an operation (`withdraw`) that always reverts. OpenZeppelin's own `ERC4626Upgradeable` documentation notes that `maxWithdraw` and `maxRedeem` must stay consistent; overriding one to `0` and the other to `balanceOf` breaks that coupling.

The practical impact is on external integrators rather than on the vault's internal accounting: a lending market, price feed, or vault-of-vaults that values sUSDat collateral via previewRedeem will value it up to 5% above the amount actually obtainable, and the discrepancy widens exactly when the protocol switches to the elevated fee tier.

Recommendations:

Consider aligning the ERC-4626 preview helpers with the vault's actual exit behaviour:

1. Have `previewWithdraw` return `0`, since synchronous `withdraw` is not supported.
2. Have `previewRedeem` apply `redemptionFeeBps`, so the quoted amount reflects the fee that would apply on the async redemption path.

Customer Response:

Fixed in 1a963b3.

Fix Review:

Fix confirmed.

L-05: ERC-165 omits IWithdrawalQueueERC721

Severity: Low	Impact: Low	Likelihood: High
Files: src/v2/WithdrawalQueueERC721.sol	Status: Fixed	

Description:

WithdrawalQueueERC721 implements IWithdrawalQueueERC721, but its supportsInterface override only forwards to the inherited ERC-721 / AccessControl path:

```
function supportsInterface(bytes4 interfaceId)
    public
    view
    override(ERC721EnumerableUpgradeable, AccessControlUpgradeable)
    returns (bool)
{
    return super.supportsInterface(interfaceId);
}
```

As a result, queries for type(IWithdrawalQueueERC721).interfaceId (0xcf58cc7c) return false, even though the contract exposes that interface's functions. Standard ERC-721 / AccessControl IDs still resolve correctly via super.

Integrators or tooling that use ERC-165 to detect the queue's protocol interface will treat a live queue as unsupported.

Recommendations:

Return true for type(IWithdrawalQueueERC721).interfaceId in supportsInterface, in addition to super.supportsInterface(interfaceId).

Customer Response:

Fixed in e428f67.

Fix Review:

Fix confirmed.

L-06: Vested surplus inflates NAV but is siloed from the redemption liquidity check, causing false InsufficientLiquidity

Severity: Low	Impact: Medium	Likelihood: Low
Files: src/v2/StakedUSDat.sol	Status: Fixed	

Description:

totalAssets() includes vested surplus (surplusVestingAmount - getUnvestedSurplus()) in NAV, and convertToAssets uses this NAV to price redemption shares. But the liquidity check uses only usdatBalance. The _sweep() call at the top of redeemQueuedShares only moves surplus into usdatBalance when 100% vested. During partial vesting (e.g., 90% vested), the vested value inflates the share price used by convertToAssets, but the liquidity gate cannot see it. Users attempting to exit at NAV during active vesting get InsufficientLiquidity even though the vault holds ample funds.

Recommendations:

Make _sweep() partial, keep the invariant that vested value lives in exactly one place. Today _sweep() is all-or-nothing:

```
function _sweep() private {
    uint256 amount = surplusVestingAmount;
    if (amount == 0 || getUnvestedSurplus() != 0) return;

    surplusVestingAmount = 0;
    usdatBalance += amount;

    emit SurplusSwept(amount);
}
```

Add one storage slot to track how much of the tranche has already been folded in, and release continuously:

```
/// @dev Portion of the active tranche already folded into `usdatBalance`.
uint256 public surplusSwept;

function _sweep() private {
    uint256 amount = surplusVestingAmount;
    if (amount == 0) return;

    uint256 unvested = getUnvestedSurplus();
    uint256 releasable = amount - unvested - surplusSwept;

    if (releasable != 0) {
        usdatBalance += releasable;
        emit SurplusSwept(releasable);
    }

    if (unvested == 0) {
        surplusVestingAmount = 0;
        surplusSwept = 0;
    } else {
        surplusSwept += releasable;
    }
}

function totalAssets() public view override returns (uint256) {
    uint256 vestedSurplus = surplusVestingAmount - getUnvestedSurplus() -
surplusSwept;
    return usdatBalance + vestedSurplus + strcMirrorModule.recognizedValue() +
strconModule.recognizedValue();
}
```

Customer Response:

Fixed in 79433c9 (also refined in PR#108).

Fix Review:

Fix confirmed.

L-07: Redemption-fee accretion raises PPS per settlement; request splitting and operator ordering decide who captures it

Severity: Low	Impact: Medium	Likelihood: Medium
Files: src/v2/StakedUSDat.sol src/v2/WithdrawalQueueERC721.sol	Status: Fixed	

Description:

Each successful redemption leaves the fee in the vault while burning the redeemed shares, so price-per-share rises for everyone still in:

```
/// The fee stays in the vault and accrues to remaining shares.
function redeemQueuedShares(uint256 shares, uint256 minSharePrice)
    ...
    uint256 gross = convertToAssets(shares);
    uint256 fee = Math.mulDiv(gross, redemptionFeeBps(), BPS_DENOMINATOR,
Math.Rounding.Ceil);
    usdat = gross - fee;
    ...
    usdatBalance -= usdat;
    _burn(address(WITHDRAWAL_QUEUE), shares);
```

processRequests then settles each ID at the live NAV after the previous fill:

```
function processRequests(uint256[] calldata tokenIds) external ...
onlyRole(OPERATOR_ROLE) {
    for (uint256 i = 0; i < count; i++) {
        ...
        (IStakedUSDat.RedemptionResult result, uint256 usdat) =
            STAKED_USDAT.redeemQueuedShares(req.shares, req.minSharePrice);
        if (result == IStakedUSDat.RedemptionResult.BelowLimit) continue;
        if (result == IStakedUSDat.RedemptionResult.InsufficientLiquidity)
continue;
        ...
```

```
    }
}
```

Two consequences follow:

1. **Request splitting.** A holder can cancel one large request and re-queue N smaller ones. When the operator processes those IDs in sequence, each earlier fill's retained fee lifts `convertToAssets` for the later ones, so the sum of net payouts exceeds a single redeem of the same total shares.
2. **Operator chosen beneficiaries.** "the operator selects and orders IDs; there is no FIFO guarantee." Whoever is processed later in a batch (or in a later batch) receives the higher PPS. A selective `OPERATOR_ROLE` holder can settle affiliated or later-created requests before earlier ones and allocate the fee accretion accordingly. `Request.timestamp` is recorded but never enforced.

Recommendations:

Process all IDs in the same batch at the same share price rather than each request at their own shareprice.

Customer Response:

Fixed in f412454.

Fix Review:

Fix confirmed.

L-08: Just-in-time deposits can capture redemption fees from queued holders

Severity: Low	Impact: Low	Likelihood: Medium

Files: src/v2/StakedUSDat.sol	Status: Acknowledged	

Description:

Redemption fees remain in usdatBalance and immediately increase the NAV of every share that survives settlement. Deposits remain open while pending requests are processed, and newly minted shares participate fully in this fee accrual. An attacker can therefore front-run a large processRequests transaction with a small Regular-mode deposit and capture a disproportionate portion of the redemption fee.

Recommendations:

Do not immediately recognize redemption fees as backing for unrestricted newly minted shares. Possible mitigations include introducing an enforced fee-vesting mechanism.

Customer Response:

Acknowledged.

L-09: setMaxRewardsBps has no upper bound, allowing unbounded reward injection

Severity: Low	Impact: Low	Likelihood: Medium
Files:	Status: Fixed	

Description:

The mirror's balance is just a number the operator attests to, and it counts fully toward the vault's value. Each reward is capped at maxRewardsBps × totalAssets(), but setMaxRewardsBps has no upper limit, so the parameter manager can raise that cap to anything. Between them, the operator and parameter manager can add unlimited fake "reward" value, pushing share price up

and letting redemptions pay out more than the vault actually holds (insolvency). The mirror is retired at migration, so this only matters during the short pre-migration window.

Recommendations:

Add a hard upper bound to setMaxRewardsBps

Customer Response:

Fixed in 2ad93be.

Fix Review:

Fix confirmed.

L-10: Blacklisted approved operators retain authority over other users' tokens and withdrawal claims

Severity: Low	Impact: Low	Likelihood: Low
Files: src/v2/StakedUSDat.sol src/v2/WithdrawalQueueERC721.sol	Status: Fixed	

Description:

Both transferFrom implementations validate only the token owner (from) and recipient (to), not msg.sender.

Consequently, an blacklisted operator that was approved can continue exercising allowances over non-blacklisted users. For sUSDat, it can transfer a victim's approved shares to an unrestricted accomplice. In the withdrawal queue, an operator restricted by either the sUSDat blacklist or USDat freeze list can similarly transfer approved request NFTs, including processed claims, between unrestricted accounts.

This defeats blacklist containment when an approved application or operator is compromised and subsequently restricted.

Recommendations:

Require the caller to be unrestricted in both transferFrom implementations:

Customer Response:

Fixed in 889f8f0 and b0b956e.

Fix Review:

Fix confirmed.

L-11: STRConModule.setOracle performs no live price smoke-test, so a rotation can immediately brick the whole vault

Severity: Low	Impact: Medium	Likelihood: Low
Files: src/v2/modules/STRCon/STRConModule.sol	Status: Fixed	

Description:

_setOracle only validates non-zero address, non-empty code and decimals() == 8: if (newOracle == address(0) || newOracle.code.length == 0) revert InvalidOracle(); ISTRConPriceOracle newPriceOracle = ISTRConPriceOracle(newOracle); if (newPriceOracle.decimals() != ORACLE_DECIMALS) revert InvalidOracle(); oracle = newPriceOracle; It never calls getPrice(). A wrapper that satisfies decimals() == 8 but reverts on getPrice() is accepted.

Because v2 is deliberately fail-closed, the very next block every value-sensitive path (totalAssets(), deposits/mints, redeemQueuedShares, buy/sell, migrate()) reverts, and maxDeposit/maxMint report 0. Recovery requires a second rotation through the 5-day PARAMETER_MANAGER_ROLE timelock, i.e. the protocol is fully halted for the duration of the delay. The same omission applies to the constructor path.

Recommendations:

Call newPriceOracle.getPrice() inside _setOracle (and optionally sanity-bound the returned value against the currently active oracle) so a non-functional wrapper is rejected atomically.

Customer Response:

Fixed in 0aeb28b.

Fix Review:

Fix confirmed.

Informational Severity Issues

I-01: setMaxApiStaleness / setDeviationBps accept 0, permanently bricking `getPrice()`

Description:

Both setters validate only an upper bound - setMaxApiStaleness rejects > MAX_API_STALENESS, setDeviationBps rejects > MAX_DEVIATION_BPS with no lower bound, so 0 is accepted. With maxApiStaleness = 0, the check block.timestamp - referenceUpdatedAt > 0 reverts unless the reference feed updated in the exact same second. With deviationBps = 0, deviation > 0 reverts on any non-zero feed difference. Either setting makes getPrice() revert on every call so totalAssets() also reverts and the whole vault halts.

Recommendations:

Enforce a sensible lower bound or reject 0 on both setters.

Customer Response:

Fixed in a67c5c8.

Fix Review:

Fix confirmed.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.