Infinifi PR 228

Security Review

Cantina Managed review by:

R0bert, Lead Security Researcher
Slowfi, Security Researcher

December 11, 2025

Contents

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

infiniFi is a self-coordinated depositor-driven system designed to tackle the challenges of duration gaps in traditional banking.

From Oct 7th to Oct 9th the Cantina team conducted a review of infinifi-contracts on commit hash f87c9a17. The team identified a total of **11** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	1	0	1
Medium Risk	0	0	0
Low Risk	3	2	1
Gas Optimizations	1	1	0
Informational	6	3	3
Total	**11**	**6**	**5**

3 Findings

3.1 High Risk

3.1.1 Loss handling ignores the smoothing queue

Severity: High Risk

Context: JCurveSmoother.sol#L125

Description: `YieldSharingV2.accrue()` works entirely off `unaccruedYield()` without first netting the matured balance sitting on `JCurveSmoother`. When someone calls `accrue()` while the smoother already has vested rewards, `_handleNegativeYield(_negativeYield)` immediately slashes lockers and, if the deficit is large, marches through the rest of the loss waterfall even though the smoother could have burned tokens to offset the hit:

```
uint256 lockingReceiptTokens = LockingController(lockingModule).totalBalance();
if (_negativeYield <= lockingReceiptTokens) {
    LockingController(lockingModule).applyLosses(_negativeYield);
    return;
}
LockingController(lockingModule).applyLosses(lockingReceiptTokens);
```

Because `JCurveSmoother.accrueAndSmooth` mints the queued rewards straight into total supply, those tokens are already exposed during a price cut in case of an important slash that reduces the price of the receipt token:

```
ReceiptToken(receiptToken).mint(address(this), yieldToSmooth);
```

If the loss propagates all the way to the oracle update, the queued balance is marked down along with everyone else:

```
uint256 totalSupply = ReceiptToken(receiptToken).totalSupply();
uint256 price = FixedPriceOracle(oracle).price();
uint256 newPrice = price.mulDivDown(totalSupply - _negativeYield, totalSupply);
FixedPriceOracle(oracle).setPrice(newPrice);
```

Only afterwards, when an operator finally runs `JCurveSmoother.distribute(true)`, those now-discounted tokens are burned:

```
ReceiptToken(receiptToken).burn(_vested);
```

That flow double-punishes lockers: they absorb the full loss up front and then see the same queue reappear as "new" profit that pays performance fees and follows the positive-yield split, while any price cut reduces the USD value realized when the smoother eventually burns. Because of this, the users suffer avoidable over-slashing and dilution whenever the negative yield is taken before the smoother is netted.

Recommendation: Let the loss path consume the smoother balance before measuring `unaccruedYield()` for example by storing the smoother address and calling `JCurveSmoother.distribute(false)` inside `accrue()` right before the yield check.

infiniFi: Acknowledged. Added a comment to explain this issue on the code base but there is not change in the logic itself. To properly fix this issue it would be required to upgrade the yield sharing contract.

Cantina Managed: Acknowledged by infiniFi team.

3.2 Low Risk

3.2.1 Governor-controlled smoother can drain queued iUSD

Severity: Low Risk

Context: JCurveSmoother.sol#L25

Description: `JCurveSmoother` inherits `CoreControlled`, so `GOVERNOR` can execute arbitrary external calls from the smoother via `emergencyAction`. During smoothing the contract mints unbacked iUSD to itself:

```
ReceiptToken(receiptToken).mint(address(this), yieldToSmooth);
```

Those balances remain ordinary ERC20 holdings until burned. A malicious or compromised `GOVERNOR` can issue an emergency action that calls `ReceiptToken.transfer` from the smoother to an arbitrary address, emptying the queue or simply mint unbacked iUSD constantly to any controlled external address.

Recommendation: Consider overriding the `JCurveSmoother.emergencyAction` function forcing it to always `revert()`.

infiniFi: Acknowledged. Governor role has already the power to grant itself the minter role or transfer the funds held elsewhere in the system, so this does not open a new attack vector.

Cantina Managed: Acknowledged by Infinifi team. However the research team still advises to improve the RAC model on future upgrades.

3.2.2 `FluidRewardsClaimer` recipient can be set to zero and changes are not emitted

Severity: Low Risk

Context: FluidRewardsClaimer.sol#L36

Description: The contract allows assigning the `recipient` variable directly without validating that it is not the zero address or emitting any event when the value changes. Since this address receives funds directly through reward claims, setting it to `address(0)` would cause the contract to send tokens to an irrecoverable destination, resulting in a permanent loss of funds. The absence of an emitted event also reduces traceability of changes to a critical configuration value.

Recommendation: Consider to introduce an internal `_setRecipient` function that rejects the zero address and emits an event whenever the recipient is updated. Use this function both in the constructor and in any external setter.

infiniFi: Fixed in commit 512a6d64.

Cantina Managed: Fix verified.

3.2.3 Avoid updating interpolation when `yieldToSmooth` is zero

Severity: Low Risk

Context: JCurveSmoother.sol#L89-L96

Description: When `yieldToSmooth == 0`, the code still resets the vest by setting:

- `point.rate = vesting() / interpolationDuration`.
- `point.lastAccrued = block.timestamp`.

Let:

- D = `interpolationDuration`.
- V = current unvested amount (what `vesting()` returns at the moment).
- t_θ = previous `lastAccrued`.
- t = now.
- Original end time: T_{end} = t_θ + D.
- Remaining time without reset: $\tau = T_{\text{end}} - t$ (clamped at ≥ 0).

If we do not reset, the remaining timeline is τ. If we do reset with `yieldToSmooth = 0`, the new schedule becomes a fresh D (because `rate := V/D` and the clock restarts at t), so the new remaining time is $\tau' = D$.

Hence the unintended stretch introduced by a zero-yield reset is:

$$\Delta = \tau' - \tau = D - (T_{\text{end}} - t) = t - t_\theta \quad (\geq 0)$$

which is exactly the elapsed time since the last accrual. In other words, every zero-yield reset lengthens the remaining vest by how long it's been since the previous accrual.

Numerical example:

- $D = 30$ days.
- At t_θ, a vest starts with $V_\theta = 300$ iUSD $\rightarrow$ `rate = 10 iUSD/day`.
- After 10 days, at time t, the remaining $V = 200$ iUSD and $\tau = 20$ days.
- If `yieldToSmooth = 0` and we reset at `t`, we set `rate = V/D = 200/30` $\approx$ `6.6667 iUSD/day` and restart the clock.
- The new remaining time is $\tau' = 30$ days.
- Stretch: $\Delta = \tau' - \tau = 30 - 20 = 10$ days (the vest slows down and ends 10 days later than it would have).

This behavior dilutes the release pace of already-smoothed rewards without adding new value.

Recommendation: Consider to early-return when `yieldToSmooth == 0`, leaving `point.rate` and `point.lastAccrued` unchanged so the existing vest completes on its original schedule.

infiniFi: Fixed in commit ddab79ef.

Cantina Managed: Fix verified.

3.3 Gas Optimization

3.3.1 Unused immutable variable `accounting`

Severity: Gas Optimization

Context: JCurveSmoother.sol#L33

Description: The contract defines an immutable variable `accounting`, but it is never referenced anywhere in the implementation. Keeping unused variables adds unnecessary bytecode size and can cause confusion about their intended purpose, especially for immutable values that cannot be modified.

Recommendation: Consider to remove the unused `accounting` variable or integrate it into the logic if it was meant to represent a dependency such as a yield accounting contract.

infiniFi: Fixed in commit 8fbf225c.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 Smoothed balance skews positive yield toward lockers

Severity: Informational

Context: YieldSharingV2.sol#L253-L268

Description: When `JCurveSmoother.accrueAndSmooth()` mints queued rewards to itself, those tokens immediately increase `ReceiptToken.totalSupply()` even though the assets remain locked on the smoother:

```
ReceiptToken(receiptToken).mint(address(this), yieldToSmooth);
```

In `_handlePositiveYield`, the split between stakers and lockers derives its "*illiquid floor*" from that inflated supply:

```
uint256 receiptTokenTotalSupply = ReceiptToken(receiptToken).totalSupply();
uint256 targetIlliquidMinimum = receiptTokenTotalSupply.mulWadDown(targetIlliquidRatio);
uint256 lockingReceiptTokens = LockingController(lockingModule).totalBalance();
if (lockingReceiptTokens < targetIlliquidMinimum) {
    lockingReceiptTokens = targetIlliquidMinimum;
}
lockingReceiptTokens = lockingReceiptTokens.mulWadDown(bondingMultiplier);
uint256 totalReceiptTokens = stakedReceiptTokens + lockingReceiptTokens;
```

Because the smoother's balance counts toward `receiptTokenTotalSupply`, the `targetIlliquidMinimum` is boosted whenever smoothing is active. That artificial increase flows into the final split:

```
uint256 stakingProfit = _positiveYield.mulDivDown(stakedReceiptTokens, totalReceiptTokens);
uint256 lockingProfit = _positiveYield - stakingProfit;
```

Locking users therefore receive a larger share of every positive accrual while the queue remains, even though the extra supply is just pending rewards and not actual locked capital.

Therefore, stakers are underpaid during smoothing intervals and lockers collect the marginal upside created by the iUSD tokens minted in the `JCurveSmoother` contract.

Recommendation: Exclude the smoother's balance from the total supply when deriving the illiquid floor. For example, subtract `ReceiptToken(receiptToken).balanceOf(address(jCurveSmoother))` from `receiptTokenTotalSupply` before computing `targetIlliquidMinimum`.

infiniFi: Acknowledged. The impact should be minimal for the expected values.

Cantina Managed: Acknowledged by infiniFi team.

3.4.2 Smoothing backlog re-emerges without timely `distribute()`

Severity: Informational

Context: JCurveSmoother.sol#L120-L133

Description: `JCurveSmoother` only burns matured rewards when `distribute()` is called. If that function is skipped for an extended period, `vesting()` simply grows with every interpolation period, and the next call, often with `_accrue = true`, burns the entire backlog in one transaction:

```
uint256 _vested = vested();
if (_vested != 0) {
    ReceiptToken(receiptToken).burn(_vested);
}
if (_accrue) {
    YieldSharingV2(yieldSharing).accrue();
}
```

Instead of smoothing, the protocol experiences a single reward burst larger than any individual accrual, precisely the behavior the queue meant to avoid.

Recommendation: Automate or enforce timely drains of the queue. For example, add a heartbeat keeper that calls `distribute(true)` on a fixed cadence.

infiniFi: Acknowledged. The backend should call distribute and accrue every 6 hours.

Cantina Managed: Acknowledged by infiniFi team.

3.4.3 Ambiguous error name for zero reward balance

Severity: Informational

Context: FluidRewardsClaimer.sol#L70

Description: The contract reverts with `InvalidFarm(_farm)` when a farm has no FLUID rewards available. This error name suggests an invalid configuration or address, but in practice it is triggered when the balance is simply zero. This can make operational debugging more difficult, as it conflates configuration issues with normal state conditions.

Recommendation: Consider to use a clearer revert reason or custom error name such as `NoRewards()` or `ZeroBalance()` to better reflect the actual cause of the revert.

infiniFi: Fixed in commit fe5be528.

Cantina Managed: Fix verified.

3.4.4 Magic number used for rUSD to USDC scaling

Severity: Informational

Context: ReservoirFarm.sol#L71

Description: `redeem(address(this), _rUsdIn / 1e12)` hardcodes the `1e12` factor for decimals conversion. This reduces readability and couples behavior to current token decimals.

Recommendation: Consider to replace `1e12` with a named constant (e.g., `RUSD_TO_USDC_SCALE = 1e12`) and use it in the calculation to avoid a magic number and clarify intent.

infiniFi: Fixed in commit 0b6051b9.

Cantina Managed: Fix verified.

3.4.5 Approval may exceed redeemed amount due to decimal truncation

Severity: Informational

Context: ReservoirFarm.sol#L69-L72

Description: `unwrap` approves `_rUsdIn` but redeems `_rUsdIn / 1e12`, which can leave a dangling allowance because of truncation to 6 decimals.

Recommendation: Consider to use `SafeERC20.safeApprove` to align the approved amount with what will actually be redeemed.

infiniFi: Fixed in commit 5708e81b.

Cantina Managed: Fix verified.

3.4.6 JCurve smoothing uses current weights; unwinders at E receive a decaying share

Severity: Informational

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

Description: When a spike is captured at time E, JCurve mints iUSD and later burns it linearly; each burn is accrued using the current reward weights. If a user begins unwinding at E, their locking weight decays over the unwind window, so their share of that smoothing tranche also decays. Attribution is to whoever remains illiquid during distribution, not to the weights at E. This is consistent with the current design but may feel unfair to users who helped generate the spike before E.

Recommendation: This design is currently a protection against other type of griefing that can make a more ”unfair” reward distribution. However this could be mitigated although the complexity may not cover the benefits.

Consider to:

- Keep interpolation duration short relative to the unwind period and document the behavior clearly.
- Freeze the global split at E (locking vs staked) for each tranche, distributing within each side by current weights.
- Optionally add a brief grace-on-unwind for active tranches to preserve pre-unwind weight for that tranche.

infiniFi: Acknowledged. Not an easy thing to deal with.

Cantina Managed: Acknowledged by Infinifi team. This was just a pure informational issue that does not represent any else than a design decision.