InfiniFi YieldSharing PR 169

Security Review

Cantina Managed review by:

R0bert, Lead Security Researcher
Cccz, Security Researcher

June 10, 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	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

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 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.

On Jun 9th the Cantina team conducted a review of infinifi-contracts on commit hash PR 169. The purpose of this review is to provide an independent security and correctness assessment of the code changes introduced in the pull request as well as the accompanying on-chain upgrade procedure.

2.1 Code base reviewed

The diff contained in PR 169 covering, among others, `YieldSharingV2.sol`, `InfiniFiGatewayV2.sol`, `Proposal_9.sol` and associated library/utility changes.

2.2 Upgrade pathway

A full dry-run of the timelocked upgrade sequence was reproduced, starting from the current main-net deployment (snapshot 22657924) and ending with the new system in place:

1. Deployment of `YieldSharingV2` and `InfiniFiGatewayV2` implementation contracts.
2. Execution of `Proposal_9`: role re-assignment, safety-buffer migration, parameter initialisation, proxy upgrade and pausing of legacy modules.
3. Post-migration validation steps defined in `Proposal_9.validate()`.

2.3 Review goals

- Detect logic errors, re-entrancy windows, improper access control, or state-collision issues arising from the upgrade.
- Verify that reward-calculation changes (epoch $\rightarrow$ 8-hour interpolation) preserve economic fairness and do not introduce dilution or DoS vectors.
- Confirm that the migration script leaves user balances (Receipt, Staked, Locked tokens) economically unchanged and that no funds are left inaccessible.

Unless otherwise stated, every finding, recommendation, or comment in the subsequent sections refers to the code and deployment sequence described above. The team identified a total of **3** issues:

Issues Found

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

3 Purpose of the PR: Addressing Dilution in InfiniFi's YieldSharing contract

In the InfiniFi protocol, dilution is an important issue that affects the rewards received by stakers. This section explores what dilution means in this context, how it manifests in the YieldSharingV1 contract and how YieldSharingV2 resolves it by shortening the reward-distribution window.

3.1 What "Dilution" Means in This Context

Dilution, in the InfiniFi protocol and in the YieldSharingV1 contract context, refers to a drop in the percentage of a fixed profit pool that an existing staker ultimately receives. This occurs because new stakers can join after profits are earned but before they are fully paid out, allowing them to claim a share of the rewards without contributing to the original profits. The result is a reduced effective yield for those who were staked during the profit-generating period.

3.2 How YieldSharingV1 Creates Dilution

In YieldSharingV1, the reward-distribution process spans a 7-day window, creating opportunities for dilution. Here's how it happens, step by step:

1. **Profit Detection**: When the keeper calls `accrue()`, the protocol calculates the profits from the last epoch. Suppose 700 iUSD was earned, and 30% is allocated to stakers. This yields:

   $$\text{stakingProfit} = 700 \times 0.3 = 210\,\text{iUSD}.$$

   The 210 iUSD is transferred to the siUSD vault via `StakedToken.depositRewards(210)`.

2. **Reward Smoothing**: The 210 iUSD is stored in `epochRewards[nextEpoch]` and distributed linearly over the next 7 days (604,800 s). Each second, approximately

   $$\frac{210}{604{,}800} \approx 0.000347\,\text{iUSD}$$

   becomes claimable. At the epoch's start, none of the 210 iUSD is claimable; it remains locked in the vault.

3. **Protocol Growth**: If the protocol grows rapidly, new stakers can join while rewards are still locked. For example, on day 2, approximately $\frac{5}{7}$ of the rewards (around 150 iUSD) remain locked. A new staker (e.g. a whale) mints iUSD and stakes it, acquiring, say, 50 % of the siUSD supply. Since the locked rewards are not included in `totalAssets()`, his deposit price excludes them. He then claims 50 % of the remaining 150 iUSD (75 iUSD), despite contributing nothing to the original 210 iUSD. Existing stakers' share drops from 100 % to 50 %, exemplifying dilution.

The 7-day distribution window exacerbates this issue: the longer the interpolation period, the more time newcomers have to join and share pending rewards, increasing dilution.

3.3 How YieldSharingV2 Fixes Dilution

YieldSharingV2 mitigates dilution by shortening the reward-distribution window from 7 days to *eight hours*. The core mechanics remain the same, but the shorter timeframe changes the outcome:

- **Shorter Distribution Window**: Profits are still split (e.g. 30 % to stakers), and rewards stream linearly, but over fewer seconds. For an 8-hour window (28,800 s), the rate is

  $$\frac{210}{28{,}800} \approx 0.00729\,\text{iUSD per second}.$$

- **Frequent Updates**: Before every stake or unstake, the gateway calls `distributeInterpolationRewards()`, transferring vested iUSD from the private escrow to the siUSD vault. This ensures the share price reflects available rewards in that block.

With an 8-hour window, essentially all rewards have streamed out long before significant new staking can occur one day later. A newcomer arriving after 24 hours finds no locked rewards left, preventing them from "free-riding" on past gains. The shorter the window, the smaller the dilution opportunity.

3.4 Numeric Example: V1 vs. V2

Comparison after 24 hours:

Scenario	V1 (7-day drip)	V2 (8-hour drip)
Profit for stakers	210 iUSD	210 iUSD
Already paid after 24 h	$\frac{24}{168} \times 210 \approx 30$ iUSD	$\frac{24}{8} = 3$ cycles $\rightarrow \sim 100\%$
Still locked after 24 h	180 iUSD	≈ 0 iUSD
New staker buys 50 % of supply	Gains 50 % of 180 iUSD = 90 iUSD	Gains 50 % of ≈ 0 iUSD = 0 iUSD
Original stakers finally get	210 - 90 = 120 iUSD	210 iUSD

Table 1: Dilution comparison between V1 and V2

In V1, a newcomer after 24 hours dilutes earlier stakers by 90 iUSD. In V2 (8-hour drip), rewards have already streamed out; the late joiner receives nothing extra, and early stakers keep the full 210 iUSD.

3.5 Summary

Under YieldSharingV1, each week's profit for siUSD stakers is released slowly over 7 days. New stakers entering during that period can claim part of the still-locked rewards, diluting early participants. YieldSharingV2 reduces the drip to 8 hours, meaning nearly all rewards are distributed well before a full day has passed. Late joiners therefore cannot siphon off yesterday's gains, preserving the full reward for existing stakers and eliminating dilution.

4 Findings

4.1 Medium Risk

4.1.1 Reset of `stakedReceiptTokenCache` on Upgrade

Severity: Medium Risk

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

Description: This issue appears during the migration from `YieldSharing` to `YieldSharingV2` as the storage value of `stakedReceiptTokenCache` is not migrated. Immediately after the upgrade both cached fields are zero:

```
struct StakedReceiptTokenCache {
    uint48  blockTimestamp;  // timestamp of last accrual
    uint208 amount;          // total staked receipt tokens
}

StakedReceiptTokenCache public stakedReceiptTokenCache; // == StakedReceiptTokenCache(0, 0)
```

When the first post-upgrade call to `accrue()` executes, the contract interprets any existing siUSD supply as initially present since the beginning of that block. A flash-loan attacker can profit from this behaviour.

Abuse scenario (single block, repeatable).

1. Take a large flash-loan in USDC.
2. Call `InfiniFiGatewayV2.mintAndStake()` to obtain a huge siUSD position.
3. Call `Timelock.execute()` to run `Proposal_9`. Now `YieldSharingV2.stakedReceiptTokenCache == StakedReceiptTokenCache(0,0)`.
4. Call `YieldSharingV2.accrue()`. The loaned balance is written into the cache and treated as if it had existed.
5. `unstake()` and repay the flash-loan. The attacker succesfully inflated the staked token/liquid side vs the locked side as in `_handlePositiveYield`:

   ```
   uint256 stakedReceiptTokens = _getCachedStakedReceiptTokens().mulWadDown(liquidReturnMultiplier);
   uint256 totalReceiptTokens = stakedReceiptTokens + lockingReceiptTokens;
   uint256 stakingProfit = _positiveYield.mulDivDown(stakedReceiptTokens, totalReceiptTokens);
   // stakedReceiptTokens will be (due to the flashloan) the 99,9% of the totalReceiptTokens amount
   ```

6. This way the attacker pulls an unfair amount of rewards to the liquid side, or which is the same, to the `siUSD` holders.

Why it is actually exploitable on main-net: The protocol uses OpenZeppelin's `TimelockController`. This contract distinguishes three roles:

- **PROPOSER_ROLE** - may schedule operations.
- **CANCELLER_ROLE** - may cancel pending operations.
- **EXECUTOR_ROLE** - may execute operations after the delay.

On the deployed instance at 0x4B174a...32 the `EXECUTOR_ROLE` is deliberately granted to the `address(0)` (the "open executor" pattern). This means:

Once the 7-day delay has elapsed, anyone can call `execute()`.

Consequences for the attack:

1. The legitimate governance multisig queues `Proposal_9` and waits seven days.
2. An attacker monitors the chain for the expiration block/time.
3. **In a single flash-loan transaction** the attacker:
   (a) Borrows USDC.
   (b) Calls `mintAndStake()`.

(c) Invokes `Timelock.execute()` themselves.

(d) Immediately calls `accrue()` and then `unstake()`.

(e) Repays the flash-loan.

4. No front-running is required as the attacker is the executor. Governance cannot prevent this because execution is permissionless once the delay is over.

Because the flash-loan, the protocol upgrade and the `accrue` call happen inside the same atomic transaction, there is zero capital risk for the attacker and no reliable way for honest users to intervene.

This risk exists because the protocol timelock has an `EXECUTOR_ROLE` that anyone can use once the 7-day delay elapses (see https://etherscan.io/address/0x4B174afbeD7b98BA01F50E36109EEE5e6d327c32#readContract).

Recommendation:

1. During the upgrade, initialize `stakedReceiptTokenCache` in `YieldSharingV2` with the current cached values from the old contract.
2. Alternatively, make the first call to `accrue()` revert unless `blkTimestamp > 0`.
3. Restrict the timelock's `EXECUTOR_ROLE` to trusted accounts to prevent front-running of the upgrade transaction.
4. Finally, the most straightforward fix given that the task is already scheduled in the `Timelock` is calling `accrue` 1 minute before the task can be executed. This way, the `unaccruedYield` will be minimal.

4.2 Low Risk

4.2.1 Potential griefing vector due to rounding

Severity: Low Risk

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

Description: This issue is reflected in the `unaccruedYield` function implemented by `YieldSharingV2` (and also by `YieldSharing`) every time a user stakes right after an `YieldSharingV2.accrue` call. This causes 'unaccruedYield' to go negative and therefore blocking any redemption.

Why does it go negative?

1. A keeper (or any user) calls `accrue()`. At this point `unaccruedYield = 0`.
2. Within the same block, an attacker calls `mintAndStake()`: The mint flow ends in `afterMint()` which immediately deposits the freshly-minted receipts into a farming adapter. Many adapters round the deposited amount down by one wei; this 1-wei shortfall is visible to `Accounting.totalAssetsValue()`.
3. The total supply of `ReceiptToken` increased by the full amount, but the reported assets are smaller by the adapter's rounding loss L.
4. The difference becomes negative in the formula above `assetsInReceiptTokens - totalSupply`, resulting in `unaccruedYield = - L`.
5. Any later `unstake()` reverts because `StakedToken.maxRedeem()` forbids operations while `unaccruedYield < 0`.

This can be abused to block redemptions over multiple blocks. A malicious user could, perform the following operations at the beginning of every block for XYZ amount of consecutive blocks:

1. Call `YieldSharingV2.accrue()`. `unaccruedYield` is now 0.
2. Call `InfiniFiGatewayV2.mintAndStake()`. `unaccruedYield` is now `-1000000000000` as:

```
emit DebugUint(a: "assetsInReceiptTokens",                    b: 1738645655818200000000000000 [1.738e25])
emit DebugUint(a: "ReceiptToken(receiptToken).totalSupply()", b: 1738645655818300000000000000 [1.738e25])
```

3. Any `InfiniFiGatewayV2.unstake()` call in that block will now revert due to the `maxRedeem` implementation in the `StakedToken` contract:

```
function maxRedeem(address _receiver) public view override returns (uint256) {
    if (paused() || YieldSharing(yieldSharing).unaccruedYield() < 0) { // <----
        return 0;
    }
    return super.maxRedeem(_receiver);
}
```

See `test_6_yieldDecreasesAfterDepositDOS` in the following gist: https://gist.github.com/r0bert-ethack/b5911142df66dc4aa732f447e5f1c7e0

While it is true that the user trying to unstake could bypass this attack vector by calling `YieldSharingV2.accrue()` himself, it is not really something we should expect from users trying to unstake.

On the other hand, unstaking after an `accrue` call will also result in the increase of `unaccruedYield` caused by the rounding error in the opposite direction:

```
emit DebugUint(a: "assetsInReceiptTokens", b: 173664565581830000000000000 [1.736e25])
emit DebugUint(a: "ReceiptToken(receiptToken).totalSupply()", b: 173664565581820000000000001 [1.736e25])
```

See `test_5_accrueDOS` in the same gist file shared.

4.2.2 Delayed application of updated `interpolationDuration`

Severity: Low Risk

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

Description: The vesting cadence in `YieldSharingV2` is driven by the `Point` struct:

```
struct Point {
    uint32  lastAccrued;  // last time `accrue()` was called
    uint32  lastClaimed;  // last time rewards were streamed
    uint208 rate;         // iUSD per second currently vesting
}
```

Inside `_handlePositiveYield` (lines 305-308) the struct is refreshed only when the function observes a strictly positive `stakingProfit`:

```
if (stakingProfit > 0) {
    StakedToken(stakedToken).depositRewards(stakingProfit);
    ReceiptToken(receiptToken).transfer(address(escrow), stakingProfit);

    // (re-)compute the streaming rate
    point.rate        = uint208(vesting() / interpolationDuration);
    point.lastAccrued = uint32(block.timestamp);
}
```

Because the `rate` and `lastAccrued` fields are updated only when a new positive profit is detected, any governance change to `interpolationDuration` (e.g. moving from 8 h to 3 h or vice-versa) remains unapplied until the next profitable epoch. If the protocol happens to generate zero or negative yield for a prolonged period, the vesting timetable continues to rely on the old duration, contrary to governance intent and user expectations.

Recommendation: In the setter that changes `interpolationDuration`, invoke `accrue()` and recompute the `Point` immediately, even if the resulting `stakingProfit` is zero.