Infrared PR 647

Security Review

Cantina Managed review by:
R0bert, Lead Security Researcher
Cryptara, Security Researcher

November 26, 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

Infrared simplifies interacting with Proof of Liquidity with liquid staking products such as iBGT and iBERA.

From Nov 11th to Nov 19th the Cantina team conducted a review of infrared-contracts on commit hash b4211e37. The team identified a total of **16** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	3	3	0
Low Risk	6	2	4
Gas Optimizations	1	1	0
Informational	5	4	1
Total	**16**	**11**	**5**

2.1 Scope

The security review had the following components in scope for infrared-contracts on commit hash b4211e37:

```
src
├── core
│   ├── InfraredV1_10.sol
│   ├── libraries
│   │   └── RewardsLib.sol
│   └── StakedIR.sol
└── periphery
    ├── BatchClaimerV2_2.sol
    ├── IRAuction.sol
    └── IRRewardDistributor.sol
```

3 Findings

3.1 High Risk

3.1.1 Carry-over rewards can block transfers

Severity: High Risk

Context: IRRewardDistributor.sol#L337-L366

Description: `IRRewardDistributor.distributeRewards` approves Infrared for only `remainingRewards`, but each vault transfer adds its local carry-over on top of the base share. Once any carry-over exists, the cumulative amount pulled via `infrared.addIncentives` exceeds the allowance and every call reverts, so the same carry-over is re-assigned and can never clear even though the contract holds enough IR.

```
if (len > 0 && remainingRewards > 0) {
    _irToken.safeApprove(address(infrared), remainingRewards);

    for (uint256 i = 0; i < len; i++) {
        address stakingToken = allVaults[i];
        uint256 weight = defaultVaultWeights[stakingToken];
        if (weight == 0) continue;

        uint256 rewardAmount =
            (remainingRewards * weight) / totalDefaultWeight;
        rewardAmount += carryOverRewards[stakingToken];

        if (rewardAmount > 0) {
            try infrared.addIncentives(
                stakingToken, address(_irToken), rewardAmount
            ) {
                carryOverRewards[stakingToken] = 0;
            } catch (bytes memory errorData) {
                carryOverRewards[stakingToken] = rewardAmount;
            }
        }
    }
}
```

Once `carryOverRewards[stakingToken]` becomes non-zero, `rewardAmount` can exceed the approved `remainingRewards`. The first `infrared.addIncentives` that pushes the cumulative transfer past the allowance reverts, the vault keeps the same carry-over and future epochs repeat the failure, permanently starving that vault of IR rewards.

Recommendation: Approve the full amount that might be transferred (base share plus total carry-over) before looping, or reset the allowance per vault to exactly `rewardAmount` so carry-over payments have sufficient allowance to succeed.

Infrared Finance: Fixed in commit 94472ef.

Cantina Managed: Fix verified.

3.2 Medium Risk

3.2.1 `StakedIR` deploy script lacks seed funding

Severity: Medium Risk

Context: StakedIR.sol#L181-L183

Description: `StakedIR.initialize` mints anti-inflation shares by calling `deposit(10 ether, address(this))`, which immediately pulls 10 IR from `msg.sender` (the initializer). When `script/UpgradeInfraredV1_10.s.sol` deploys the upgrade peripherals it routes the CREATE2 proxy deployment through `ProxyDeployer.deployProxyCreate2`. During initialization the `msg.sender` is therefore the `ProxyDeployer` contract, but the script never transfers IR into that contract. The only approval executed before deployment is

`ERC20(_irToken).approve(predictedAddress, 10 ether);`, which approves the new proxy to take tokens from the broadcaster EOA instead of from `ProxyDeployer`. Because `ProxyDeployer` has neither balance nor allowance, the initializer's `safeTransferFrom` reverts and `StakedIR` can not be deployed, halting the entire upgrade automation. This bricks the scripted upgrade path and forces manual intervention.

Recommendation: Before calling `deployProxyCreate2`, send the 10 IR seed amount to the `ProxyDeployer` contract (for example via `IERC20(_irToken).transfer(address(deployer), 10 ether)`) and have the deployer approve the `StakedIR` proxy, or modify `ProxyDeployer` so it temporarily holds the funding and grants the allowance itself. Ensuring the deployer owns the seed IR allows `StakedIR.initialize` to complete successfully and keeps the scripted upgrade fully automated.

Infrared Finance: Fixed in e1b9856

Cantina Managed: Fix verified.

3.2.2 Potential revert in `sweepPayoutToken` function

Severity: Medium Risk

Context: IRAuction.sol#L145-L153

Description: `IRAuction.sweepPayoutToken` reads the IR balance but ignores it, always approving and compounding exactly `payoutAmount`. If the contract holds less than `payoutAmount` IR the `SIR_VAULT.compound(payoutAmount)` call reverts, leaving small residual balances stuck. When the contract holds more than `payoutAmount`, each sweep only processes that fixed chunk, so multiple calls are required to drain the balance and any griefing deposit just under `payoutAmount` deadlocks the workflow until governance intervenes.

```
function sweepPayoutToken() external {
    uint256 balance = IR_TOKEN.balanceOf(address(this));
    if (balance == 0) revert Errors.InsufficientBalance();
    // Approve sIR vault to spend IR for compounding
    IR_TOKEN.safeApprove(address(SIR_VAULT), payoutAmount);

    // Compound into sIR vault (increases share value for all holders)
    SIR_VAULT.compound(payoutAmount);
}
```

`balance` is used solely for the zero-balance guard, so any `balance < payoutAmount` leads to a permanent revert and any `balance > payoutAmount` leaves leftover IR that can only be cleared by repeated sweeps or emergency recovery, stranding capital in the auction contract.

Recommendation: Approve and compound the actual balance (or a caller-provided amount clamped to it) so the sweep always processes the available IR.

Infrared Finance: Fixed in commit f2b1598b.

Cantina Managed: Fix verified.

3.2.3 Stale weights skew reward math

Severity: Medium Risk

Context: IRRewardDistributor.sol#L236-L263

Description: `updateDefaultWeights` only overwrites `defaultVaultWeights` for the staking tokens provided in the call but still overwrites `totalDefaultWeight` with the sum of just those inputs. Eligible vaults omitted from the update keep their previous numerator while the denominator now reflects only the subset that was reconfigured. When `distributeRewards` iterates every eligible vault, their stale weights are still used, so the aggregate numerator can exceed the denominator. The loop therefore attempts to transfer more IR than the `remainingRewards` allowance and `infrared.addIncentives` reverts, causing the vault being processed to fall into permanent carry-over even though the keeper simply meant to change a different vault's weight.

```
function updateDefaultWeights(
    address[] calldata stakingTokens_,
    uint256[] calldata weights
) external virtual onlyKeeper {
    if (stakingTokens_.length != weights.length) {
        revert InvalidArrayLength();
    }

    address ibgt = address(infrared.ibgt());
    uint256 newTotalWeight = 0;
    for (uint256 i = 0; i < stakingTokens_.length; i++) {
        address stakingToken = stakingTokens_[i];
        if (stakingToken == address(0)) revert InvalidVault();
        if (stakingToken == ibgt) revert InvalidVault();
        if (!_eligibleVaults.contains(stakingToken)) revert InvalidVault();
        defaultVaultWeights[stakingToken] = weights[i];
        newTotalWeight += weights[i];
        emit WeightsUpdated(stakingToken, weights[i]);
    }

    if (newTotalWeight != BASIS_POINTS) revert InvalidWeights();
    totalDefaultWeight = newTotalWeight;
}
```

Any vault excluded from `stakingTokens_` retains its old `defaultVaultWeights[...]`, so the future numerator includes those stale entries while `totalDefaultWeight` equals only the sum of the fresh subset, making the reward split inconsistent and ultimately reverting mid-loop.

Recommendation: Require the caller to supply weights for every eligible vault in one transaction (setting omitted vaults to zero) or recompute `totalDefaultWeight` by iterating the entire `_eligibleVaults` set after writes so the denominator always reflects the same population as the numerators. Clearing stale entries before accepting the new totals also resolves the mismatch.

Infrared Finance: Fixed in commit ba2f756e.

Cantina Managed: Fix verified. `updateDefaultWeights` now reverts unless `stakingTokens_.length == _eligibleVaults.length` and tests were updated to pass all eligible vaults. That resolves the original omission bug when callers supply the full unique set. A minor residual edge case remains: The function still allows duplicate entries, so a caller could satisfy the length check while leaving one vault's stale weight in place. If you want to also address that, enforce uniqueness or rebuild weights from `_eligibleVaults`.

3.3 Low Risk

3.3.1 Claimable view lags by 1 block

Severity: Low Risk

Context: StakedIR.sol#L429-L444

Description: `StakedIR.getQueuedAmount` reverses through withdrawal tickets and breaks only when it encounters `ticket.unlockTime < block.timestamp`, so tickets whose unlock time equals the current timestamp remain counted as ”queued”. However `_claimWithdraw` and `claimWithdraw` allow `block.timestamp >= unlockTime`, meaning those tickets are already claimable. Because `getClaimableAmount` is defined as `totalReserved - getQueuedAmount`, it reports zero claimable assets during the exact block when a ticket matures, which can confuse automation or monitoring even though the user can successfully claim.

```
for (uint256 i = len; i > 0; i--) {
    WithdrawalRequest memory ticket = $.requests[i];
    // break loop if unlock time hsa passed
    if (ticket.unlockTime < block.timestamp) break;
    queuedAmount += ticket.assets;
}
```

A request that unlocks this block satisfies `block.timestamp >= ticket.unlockTime` (so `claimWithdraw` succeeds) but still contributes to `queuedAmount`, making `getClaimableAmount` under-report by that amount until the next block.

Recommendation: Treat tickets with `unlockTime <= block.timestamp` as claimable by changing the loop guard to `if (ticket.unlockTime <= block.timestamp) break;` (or by summing queued amounts only while `ticket.unlockTime > block.timestamp`) so the view reflects reality immediately when unlock thresholds are met.

Infrared Finance: Fixed in commit 9ef7c930.

Cantina Managed: Fix verified.

3.3.2 Operator harvest always logs 0

Severity: Low Risk

Context: RewardsLib.sol#L79

Description: `RewardsLib.harvestOperatorRewards` returns `_amt` but never assigns it, so the caller always receives zero even when iBERA was swept and `notifyRewardAmount(amountOperators)` succeeded. `InfraredV1_10.harvestOperatorRewards` forwards this value into the `OperatorRewardsDistributed` event, meaning on-chain accounting and automation always observe `amount=0` despite real rewards being paid.

```
function harvestOperatorRewards(
    RewardsStorage storage $,
    address ibera,
    address distributor
) external returns (uint256 _amt) {
    // ...
    (amountOperators, amountProtocol) =
        chargedFeesOnRewards(iBERAShares, feeTotal);
    // ...
    if (amountOperators > 0) {
        ERC20(ibera).safeApprove(distributor, amountOperators);
        IInfraredDistributor(distributor).notifyRewardAmount(
            amountOperators
        );
    }
}
```

Because `_amt` is never set to `amountOperators`, every call returns zero and the protocol emits misleading data about operator payouts.

Recommendation: Assign `_amt = amountOperators;` (and optionally zero it when the function early-returns) before exiting so callers and events reflect the actual reward amount distributed to operators.

Infrared Finance: Fixed in commit 9c2e17b8.

Cantina Managed: Fix verified.

3.3.3 Fixed-price `IRAuction` drains backlog

Severity: Low Risk

Context: IRAuction.sol#L92-L97

Description: `IRAuction.claimFees` charges a constant `payoutAmount` of IR no matter how many reward tokens/amounts the caller withdraws:

```
uint256 senderBalance = IR_TOKEN.balanceOf(msg.sender);
if (senderBalance < payoutAmount) revert Errors.InsufficientBalance();
IR_TOKEN.safeTransferFrom(msg.sender, address(this), payoutAmount);
// ...
for (uint256 i = 0; i < rewardTokens.length; i++) {
    ERC20 _token = ERC20(rewardTokens[i]);
    uint256 _amount = amounts[i];
    // ...
    _token.safeTransfer(_recipient, _amount);
}
```

Keepers can let WBERA/iBGT/IR rewards accumulate in `RewardsLib.collectBribes*` and then claim the entire backlog by passing arrays containing every token/balance while paying only that fixed `payoutAmount`. Only the constant IR fee is compounded into sIR, so the rest of the value that should have gone to sIR holders is captured by the caller. There is no per-token pricing, auction, or pro-rata check.

Recommendation: Tie the IR payment to the value withdrawn (e.g., per-token pricing, Dutch auctions, or per-deposit bids) or constrain each claim to a single reward deposit priced when it enters the auction. Otherwise a fixed payout allows keepers to drain arbitrary reward baskets at a discount.

Infrared Finance: Acknowledged. Presently, our team are the only keepers, so we make sure this is not the case. This is true for `IRAuction`, `BribeCollector` and `HarvestBaseCollector`. We are looking at making the auctions more sophisticated, by, for example, using Pyth oracles e.g. (WrappedVaultOracleAuction.sol). Downside to oracle style is that we would need to run a network of Pyth oracle updaters to keep prices fresh.

Cantina Managed: Acknowledged.

3.3.4 `StakedIR` compounding front-runnable

Severity: Low Risk

Context: IRAuction.sol#L97-L143

Description: `IRAuction.claimFees` pays a flat `payoutAmount` of IR and immediately calls `SIR_VAULT.compound(payoutAmount)`. `StakedIR.compound` just transfers the IR in and increases `totalAssets` without minting or locking shares. Because anyone can call the ERC4626 `deposit/redeem` routes, a bot can front-run a pending auction claim with a large deposit, let the compound happen so the share price jumps and then immediately queue a withdrawal via `redeem` to lock in the higher `previewRedeem` amount. The only delay is the fixed 7day withdrawal period, so opportunistic capital can repeatedly cycle in for one period and siphon most of every compounding event at the expense of long-term sIR holders.

Recommendation: Consider implementing some sort of ”rewards epoch smoothing”: lock each comp event into a reward bucket and vest it linearly over the withdrawal delay (7 days) so the share price accretes gradually and newcomers only capture the portion proportional to time held, similar to Ethena's USDe drip.

Infrared Finance: Acknowledged.

Cantina Managed: Acknowledged.

3.3.5 Permissionless auto-unwrapping in `batchClaim`

Severity: Low Risk

Context: BatchClaimerV2_2.sol#L101-L115

Description: `BatchClaimerV2_2.batchClaim` is callable by anyone, and after harvesting rewards it attempts to redeem the user's entire wBYUSD and wiBGT balances whenever this contract has allowance. If a user previously approved the claimer (e.g., with `type(uint256).max`), any third party can call `batchClaim(user, ...)` and trigger `wBYUSD.redeem(balance, user, user)` and `wiBGT.redeem(balance, user, user)` at arbitrary times. This forces the user out of wrapped positions, potentially causing exit fees, breaking compounding strategies, or ruining tax planning, even though the funds ultimately return to the user.

Recommendation: Require `msg.sender == user` (or a signed permit / opt-in flag) before invoking the auto-unwrapping block, or let users disable that feature per call, so third parties cannot force wrappers to unwind without consent.

Infrared Finance: Acknowledged. We do not see this as an issue because wBYUSD and wiBGT are 1:1 wrappers that are not of any use to a user so underlying token is more useful.

Cantina Managed: Acknowledged.

3.3.6 Incorrect Assumption of Monotonically Increasing Unlock Times

Severity: Low Risk

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

Description: The `getQueuedAmount` function assumes that each successive withdrawal request has an unlock time later than the previous one. Because the loop iterates backward and stops as soon as it finds a ticket whose unlock time has already passed, this only works if unlock times are strictly increasing.

When governance reduces the withdrawal period via `setWithdrawalPeriod`, a new request can end up with an earlier unlock time than older ones. This breaks the chronological ordering and causes the loop to stop too early.

Proof of Concept: Example illustrating the issue:

- At time 1000, Request 1 is created with an unlock time of 1500.
- At time 1200, governance reduces the withdrawal period from 500s to 200s.
- At time 1200, Request 2 is created with an unlock time of 1400.

Now the queue looks like:

- Request 1 $\rightarrow$ unlock 1500 (still locked at time 1450).
- Request 2 $\rightarrow$ unlock 1400 (already unlocked at time 1450).

When calling `getQueuedAmount()` at time 1450, the loop starts from the newest request:

1. It checks Request 2 (unlock 1400). Since `1400 < 1450`, the loop breaks immediately, assuming all older requests are also unlocked.
2. As a result, the function never reaches Request 1, even though its unlock time (1500) is still in the future.
3. The reported queued amount is 0, even though 100 assets from Request 1 remain locked.

This produces a temporarily incorrect queued total until Request 1 also unlocks.

Recommendation: If precise queue totals are required in future upgrades, the vault should avoid assuming chronological ordering and instead scan all pending requests, index them, or enforce monotonic unlock times. If the current behavior is acceptable, external documentation should clearly state that `getQueuedAmount` may return incomplete values after a withdrawal period reduction.

Infrared Finance: Acknowledged. We will make sure that when reducing the withdrawal period the resulting `now + period` is not lower than the last requested withdrawal unlock time.

Cantina Managed: Acknowledged.

3.4 Gas Optimization

3.4.1 Cache distributor token references

Severity: Gas Optimization

Context: IRRewardDistributor.sol#L291

Description: `IRRewardDistributor.distributeRewards` repeatedly reads `_irToken` and `infrared` from storage inside the vault loop. Every `safeApprove` and `infrared.addIncentives` call triggers new SLOADs even though the addresses never change during a distribution.

```
if (len > 0 && remainingRewards > 0) {
    _irToken.safeApprove(address(infrared), remainingRewards);

    for (uint256 i = 0; i < len; i++) {
        // ...
        if (rewardAmount > 0) {
            try infrared.addIncentives(
                stakingToken, address(_irToken), rewardAmount
            ) { ... }
```

This wastes two warm SLOADs per vault iteration (and again at the top-level approval), so gas scales unnecessarily with the number of eligible vaults even though the contract could just reuse cached addresses.

Recommendation: Cache the immutable references once per call, e.g. `ERC20 irToken = _irToken; IInfrared infra = infrared;` (or just the raw addresses) and use those locals for the `safeApprove/addIncentives` calls, avoiding redundant SLOADs as the loop iterates.

Infrared Finance: Fixed in commit 44391a67.

Cantina Managed: Fix verified.

3.5 Informational

3.5.1 `StakedIR` pause has no effect

Severity: Informational

Context: StakedIR.sol#L22-L25

Description: `StakedIR` inherits `Upgradeable`, which already mixes in `PausableUpgradeable`, and initialization grants the `PAUSER_ROLE`. However, none of the contract's state-changing entry points are gated by `whenNotPaused` (or any equivalent). Functions like `compound`, the withdrawal queue helpers (`_requestWithdraw`, `withdraw`, `redeem`), and the claim functions (`claimWithdraw`, `claimBatch`) execute unconditionally, so calling `pause()` never blocks deposits, withdrawals, or compounding despite the advertised circuit breaker.

```
function compound(uint256 amount) external virtual nonReentrant {
    if (amount == 0) revert Errors.ZeroAmount();
    // ...
    IERC20(asset()).safeTransferFrom(msg.sender, address(this), amount);
    $.totalCompounded += amount;
}

function withdraw(uint256 assets, address receiver, address owner)
    public
    virtual
    override
    returns (uint256 shares)
{
    shares = previewWithdraw(assets);
    _requestWithdraw(shares, receiver, owner);
}
```

Since none of these entry points check the paused state, even emergency calls to `pause()` leave every operation fully functional, preventing governance from halting flows during an incident.

Recommendation: Consider adding the `whenNotPaused` modifier to the `deposit/mint` functions and to `_requestWithdraw` (which `withdraw/redeem` call). Existing queued withdrawals could still be claimed via `claimWithdraw` even while paused. Document that pause is reserved for emergencies (e.g., detected accounting bug) so users understand it's a ”stop new flows” safeguard.

Infrared Finance: Fixed in commit 039870ec.

Cantina Managed: Fix verified.

3.5.2 Unused imports and event declarations

Severity: Informational

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

Description: Several source files pull in interfaces or declare events that are never referenced, adding noise and small deployment overhead:

- `IRRewardDistributor.sol` imports `IInfrared` from `src/depreciated/interfaces/IInfrared.sol` but the contract never touches that interface.

- `RewardsLib.sol` declares `event ErrorMisconfiguredIRMinting(uint256 amount);` yet nothing ever emits it.
- `RewardsLib.sol` also imports `IReward`, `IVoter`, and `IInfraredGovernanceToken` without using them anywhere in the library.

Leaving these unused references in place increases bytecode size, confuses future maintainers about intended dependencies, and can mask dead code that should be removed entirely.

Recommendation: Delete the unused imports and event declaration so each contract's dependency list reflects reality and bytecode stays minimal. If the event/imports were intended for future features, leave a TODO with context instead of shipping dead code.

Infrared Finance: Fixed in commits bf32a93f, 0f0a113e and 8670f74e.

Cantina Managed: Fix verified.

3.5.3 Typo in `StakedIR` comment and outdated `InfraredV1_9` references

Severity: Informational

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

Description: In the `StakedIR` contract there is a comment reading ”break loop if unlock time hsa passed” which contains a typo and can cause minor confusion when reading the unlock/vesting loop logic. Additionally, after the protocol upgrade all references to `InfraredV1_9` still exist in the codebase and documentation; these should point to `InfraredV1_10` to reflect the current deployed version and avoid deployment, testing, or audit confusion.

Recommendation: Correct the comment in `StakedIR` to ”break loop if unlock time has passed” and run a repository-wide update to replace `InfraredV1_9` references with `InfraredV1_10`. Optionally enforce these checks through a CI lint or spellcheck and include a version-reference update step in the upgrade checklist so future upgrades do not leave stale identifiers.

Infrared Finance: Fixed in commit 44391a67.

Cantina Managed: Fix verified.

3.5.4 Unnecessary distinction between `IR_TOKEN` and `payoutToken` storage variables

Severity: Informational

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

Description: In `IRAuction` both an ERC20-typed variable `IR_TOKEN` and an address-typed `payoutToken` are stored and initialized to the same value in initialize(`_irToken`). The contract uses `IR_TOKEN` for token operations (`balanceOf`, `safeTransferFrom`, `safeApprove`) while `payoutToken` is used only for equality checks (e.g., in `claimFees` to prevent paying out the fee token). Keeping both creates redundant state, increases storage footprint by one slot, and introduces a risk of divergence if one value is ever changed independently in future upgrades or maintenance.

Recommendation: Consolidate to a single source of truth for the IR payout token to eliminate redundancy and reduce the chance of inconsistent values.

Infrared Finance: Fixed in commit 44391a67.

Cantina Managed: Fix verified.

3.5.5 Missing `maxWithdraw` and `maxRedeem` Enforcement

Severity: Informational

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

Description: The `StakedIR` contract overrides `withdraw` and `redeem` without enforcing the limits defined by `maxWithdraw` and `maxRedeem`. Under ERC-4626, these view functions are meant to express the maximum amount a user can withdraw or redeem at any given time, but in this implementation the functions are bypassed entirely.

This is intentional today and matches the queue-based withdrawal design, but it creates an upgrade-risk: future versions of the vault, frontends, or integrations may rely on these limits for liquidity controls, user throttling, or safety checks. If the max functions are later modified to return restricted values without restoring enforcement, the protocol would silently allow actions that contradict the reported limits.

Recommendation: Future upgrades that introduce any withdrawal limits should re-enable enforcement inside `withdraw` and `redeem` at the same time the max functions begin returning constrained values. Another option is to explicitly commit to non-binding max functions and document that behavior for integrators. Either approach should maintain consistency so external systems can rely on the reported limits.

Infrared Finance: Acknowledged.

Cantina Managed: Acknowledged.