Infrared PR 652

Security Review

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

December 3, 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 30th to Dec 1st the Cantina team conducted a review of infrared-contracts on commit hash 1aff9d8c. The team identified a total of **6** issues:

Issues Found

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

2.1 Scope

The security review had the following components in scope for infrared-contracts on commit hash 1aff9d8c:

```
├── script
│   └── upgrades
│       └── staking
│           └── UpgradeInfraredBERAV2_1.s.sol
└── src
    └── staking
        └── InfraredBERAV2_1.sol
```

3 Findings

3.1 High Risk

3.1.1 Pending expiry lets EL overstate stake vs CL

Severity: High Risk

Context: InfraredBERAV2_1.sol#L554-L555

Description: `InfraredBERAWithdrawor.totalPendingWithdrawals` prunes pending entries after ~256 epochs (`expiryBlock`) and shrinks `pendingWithdrawalsLength`:

```
/// @dev Internal function to get total pending withdrawals over all validators and
 ↪  remove outdated ones
function totalPendingWithdrawals() internal returns (uint256 total) {
    // clean expired pending and total valid
    uint256 len = pendingWithdrawalsLength;
    uint256 currentBlock = block.number;
    uint256 writeIndex = 0; // Tracks position for non-expired entries

    // Iterate through all pending
    for (uint256 i = 0; i < len; i++) {
        PendingWithdrawal memory pending = pendingWithdrawals[i];
        if (uint256(pending.expiryBlock) >= currentBlock) {
            // Add to total
            total += uint256(pending.amount);
            //  keep non-expired withdrawal
            if (writeIndex != i) {
                pendingWithdrawals[writeIndex] = pending;
            }
            writeIndex++;
        }
    }

    // Update queue length to reflect only non-expired withdrawals
    pendingWithdrawalsLength = writeIndex;

    // Clear remaining slots (optional, for safety)
    for (uint256 i = writeIndex; i < len; i++) {
        delete pendingWithdrawals[i];
    }
}
```

Later, a keeper can call `InfraredBERAV2_1.registerViaProofs`, which computes `_pending` using the trimmed queue (`getTotalPendingWithdrawals`) and then sets `_stakes[_pubkeyHash] = _balance - _pending`. If the CL still has the withdrawal request in its pending queue when the local entry expires, `_pending` becomes 0 and the EL re-syncs to the full CL balance, forgetting the in-flight withdrawal as the CL balance is not decreased when the withdrawal request tx hits the precompile, it is decreased only when the withdrawal is processed in a future block. Because of this, the EL stake becomes higher than the CL stake. Subsequent executes can debit again while the CL drops them, widening the divergence. When the original withdrawal eventually processes on CL, EL stays overstated, enabling over-withdrawals until another reconciliation.

Recommendation: Guarantee `totalPendingWithdrawals` is accurate with CL before calling `registerViaProofs`. For example, only reconcile when you have proof the CL pending queue matches local pending (or after verifying it explicitly), so `_pending` is not zeroed while CL still enforces the withdrawal.

Infrared Finance: Fixed in `bera-proofs` PR 14 and `infrared-contracts` commit 7d4a549.

Cantina Managed: Fix verified.

3.1.2 Recovery DoS When Pending Exceeds CL Balance

Severity: High Risk

Context: InfraredBERAV2_1.sol#L554-L555

Description: `registerViaProofs` writes `_stakes[_pubkeyHash] = _balance - _pending` without guarding against `_pending > _balance`. If the CL balance drops below the locally tracked pending (e.g., slashing/fees while a large pending withdrawal is open), the subtraction underflows and reverts.

Thereafter every recovery attempt reverts, leaving the validator unrecoverable and withdrawals blocked; internal accounting cannot be resynced without code changes or manual intervention.

Recomendation: Cap or validate pending against CL balance before subtraction (e.g., use `min(_pending, _balance)` or handle the shortfall explicitly), reverting with a controlled error only when truly unrecoverable. Ensure reconciliation can progress when balance shrinks and updates deposits/exit flags consistently.

Infrared Finance: Fixed in `bera-proofs` PR 14 and `infrared-contracts` commit 7d4a549.

Cantina Managed: Fix verified.

3.2 Medium Risk

3.2.1 Execution Layer debits on inactive/just-exited validator proof

Severity: Medium Risk

Context: InfraredBERAWithdrawor.sol#L283-L287

Description: `InfraredBERAWithdrawor.execute` does not verify that the validator is currently active, it only checks the proof's `exitEpoch != type(uint64).max` and proof staleness. The consensus layer, however, enforces `validator.IsActive(currentEpoch)` before accepting a partial/full withdrawal. If a keeper submits a request using a proof for a validator that is pre-activation, or a validator that was exited/slashed between proof capture and block inclusion, the CL will drop the request but the EL will still register the full debit, creating an EL<CL stake divergence.

CL active-state gate:

```
// beacon-kit/state-transition/core/state_processor_withdrawals.go
if !validator.IsActive(currentEpoch) {
    return errors.New("validator is not active")
}
if validator.GetExitEpoch() != constants.FarFutureEpoch {
    return errors.New("withdrawal already initiated")
}
```

EL checks omit activation and only exclude explicit exits in the proof:

```
// src/staking/InfraredBERAWithdrawor.sol
// Verify validator has not been exited
if (validator.exitEpoch != type(uint64).max) {
    revert Errors.AlreadyExited();
}
// ...
// register update to stake regardless of CL acceptance
IInfraredBERAV2(InfraredBERA).register(pubkey, -int256(amount));
```

Realistic scenario: Validator has deposited but not yet activated. Keeper uses the latest beacon proof and calls `execute` for a 10k BERA partial withdrawal. EL passes its checks and debits 10k from internal stake. CL rejects with "validator is not active," enqueuing nothing. EL stake is now 10k below CL and repeated attempts can continue to drain EL accounting while the validator remains intact on CL until activation.

Timing note: On Berachain mainnet, activations are two epochs after meeting `MinActivationBalance` (250k BERA). With `SlotsPerEpoch = 192` (2s slots), that's roughly 12.8 minutes from threshold to active. Any `execute` submitted in that window, or after an exit is set but before the keeper refreshes proofs, is exposed to this mismatch.

Recommendation: Add an activation-state check to `execute` using the proofed validator state (e.g., `activationEpoch <= currentEpoch < exitEpoch`) before calling the precompile, and/or include the

current slot/epoch in the freshness check to ensure the validator is active at submission time. If the validator is pre-activation or already exited/slashed per the header, revert instead of registering a debit.

Infrared Finance: Acknowledged. We were not planning on upgrading withdrawor at this point unless it becomes critical. This scenario would not occur with our current operations because exits are manual and partial withdrawals can only occur with validators of stake > 500k but we have taken notes for future upgrade improvements on issue 645.

Cantina Managed: Acknowledged.

3.2.2 Stuck Recovery on CL-Rejected Exit

Severity: Medium Risk

Context: InfraredBERAV2_1.sol#L516-L554

Description: `registerViaProofs` returns immediately when `stake + _pending == _balance`. After a withdrawal attempt that the consensus layer rejects (e.g., pending CL withdrawal, inactive validator), the withdrawor zeroes EL stake and records a pending equal to the CL balance. In that state `stake + pending` matches the proofed CL balance, so `registerViaProofs` hits the early return and never runs the reconciliation logic that would restore `_stakes` and clear `_exited`.

The validator remains marked exited with zero internal stake until the pending entry ages out (~256 epochs), blocking exits/withdrawals and further recovery.

Recommendation: Remove or narrow the early return so recovery still runs when _exited is true or stake is zero. Reconcile `_stakes` to the proofed balance (minus validated pending) and clear `_exited` when CL shows balance, even if `stake + pending == balance`.

Infrared Finance: Fixed in `bera-proofs` PR 14 and `infrared-contracts` commit 7d4a549.

Cantina Managed: Fix verified.

3.3 Low Risk

3.3.1 RegisterViaProofs does not update deposits state variable

Severity: Low Risk

Context: InfraredBERAV2.sol#L569-L572

Description: The new `registerViaProofs` in `src/staking/InfraredBERAV2_1.sol` restores per-validator `_stakes` from consensus proofs but no longer restores `deposits` when the consensus-layer (CL) balance exceeds local accounting. In the prior V2 logic, when `_balance > stake + _pending`, `deposits` was incremented by the delta. In V2_1 this top-up is removed. For operator rebalances executed directly on `withdrawor` (no `burn/_withdraw`), this omission is benign because `deposits` was never decremented. However, CL balances can rise independently of EL accounting: beacon/CL deposits are permissionless, and a third party can top up an Infrared-managed validator directly ("bypass beacon deposits" noted in your NatSpec comments). In that case `_balance` exceeds `stake + _pending`, `registerViaProofs` updates `_stakes` but leaves `deposits` unchanged, so `totalAssets()` and mint/burn pricing remain too low relative to real CL-held funds. New minters then acquire shares against an undercounted denominator, gaining excess claim and diluting existing holders.

Recommendation: Gate the reconciliation on an explicit operator flag or a branch: when the intent is a CL top-up or accounting recovery and proofs show `_balance > stake + _pending`, increase `deposits` by the delta (or resync `deposits` to the proof-backed balance minus pending) before updating `_stakes`. Keep the current behavior (no change to `deposits`) for operator-triggered rebalances that never decremented `deposits`.

Infrared Finance: Acknowledged.

Cantina Managed: Acknowledged.

3.4 Informational

3.4.1 InfraredBERAKeeper.queueExitRebalance check can not ensure the CL has finished processing earlier requests

Severity: Informational

Context: InfraredBERAKeeper.s.sol#L69-L73

Description: The `queueExitRebalance` guard only rejects when the withdrawor's local `pendingWithdrawals` for a pubkey is non-zero. Consensus scheduling can legitimately keep a partial withdrawal pending even after the withdrawor's local queue has expired or been cleared, so this check can not ensure the CL has finished processing earlier requests. A keeper can see "no local pending" and submit another exit/rebalance; the CL may still consider a pending partial and will skip/reject the new request while EL accounting proceeds, recreating the EL<CL divergence.

CL behaviors that delay or drop processing:

1. Withdrawable epoch delay: a pending partial sets a future epoch before it can be included:

```
// beacon-kit/state-transition/core/state_processor_withdrawals.go
toWithdraw := min(balance-minActivationBalance-pendingBalanceToWithdraw,
↪  req.Amount)
currentEpoch, _ := st.GetEpoch()
nextEpoch := currentEpoch + 1
withdrawableEpoch := nextEpoch + sp.cs.MinValidatorWithdrawabilityDelay()
ppWithdrawal := &ctypes.PendingPartialWithdrawal{
    ValidatorIndex:    index,
    Amount:            toWithdraw,
    WithdrawableEpoch: withdrawableEpoch,
}
pendingWithdrawals = append(pendingWithdrawals, ppWithdrawal)
```

   The CL will not process the partial before `withdrawableEpoch`.

2. Per-payload limit with EVM inflation reservation: only a bounded number of withdrawals fit per block, and slot 0 is reserved:

```
// validateStatelessPayload
withdrawals := payload.GetWithdrawals()
if uint64(len(withdrawals)) > sp.cs.MaxWithdrawalsPerPayload() {
    return errors.Wrapf(ErrExceedMaximumWithdrawals, ...)
}
// processWithdrawals enforces first withdrawal is EVM inflation
if !payloadWithdrawals[0].Equals(st.EVMInflationWithdrawal(blk.GetTimestamp())) {
    return ErrFirstWithdrawalNotEVMInflation
}
```

   Excess eligible withdrawals are deferred to later blocks.

3. Sweep progression: the round-robin index advances per block; a validator may not be selected immediately:

```
// processWithdrawals
var nextValidatorIndex math.ValidatorIndex
if uint64(numWithdrawals) == sp.cs.MaxWithdrawalsPerPayload() {
    nextValidatorIndex =
    ↪  (expectedWithdrawals[numWithdrawals-1].GetValidatorIndex() + 1) %
    ↪  totalValidators
} else {
    nextValidatorIndex, _ = st.GetNextWithdrawalValidatorIndex()
    nextValidatorIndex += sp.cs.MaxValidatorsPerWithdrawalsSweep()
    nextValidatorIndex %= totalValidators
}
_ = st.SetNextWithdrawalValidatorIndex(nextValidatorIndex)
```

   The sweep may take multiple blocks/epochs to cycle back to a given validator.

4. Queue fullness guard: CL drops new partials when the global limit is hit (very unlikely to occur as `PendingPartialWithdrawalsLimit` is 2^27 = 134217728):

```
// processWithdrawalRequest
pendingPartialWithdrawals, _ := st.GetPendingPartialWithdrawals()
if len(pendingPartialWithdrawals) == constants.PendingPartialWithdrawalsLimit &&
↪  !isFullExitRequest {
    sp.logger.Warn("skipping processing ... queue is full", ...)
    sp.metrics.incrementPartialWithdrawalRequestDropped()
    return nil
}
```

   When saturated, new partials are ignored until the backlog is reduced.

Recommendation: Do not rely on the withdrawor's local pending map to gate exits/rebalances. Before queuing a new withdrawal, verify CL state ensuring the validator is active, not exited and that there are no pending withdrawals at the CL level.

Infrared Finance: Fixed in `bera-proofs` PR 14 and `infrared-contracts` commit 7d4a549.

Cantina Managed: Fix verified.