# Infrared contracts

## Security Review


Cantina Managed review by:

**R0bert**, Lead Security Researcher
**Cryptara**, Security Researcher



July 1, 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

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

From Jun 19th to Jun 26th the Cantina team conducted a review of infrared-contracts[feat/ibera-wthdrawals] on commit hash d24c7058. The team identified a total of **25** issues:

**Issues Found**

| Severity | Count | Fixed | Acknowledged |
|---|---|---|---|
| Critical Risk | 0 | 0 | 0 |
| High Risk | 1 | 1 | 0 |
| Medium Risk | 4 | 4 | 0 |
| Low Risk | 5 | 3 | 2 |
| Gas Optimizations | 0 | 0 | 0 |
| Informational | 15 | 4 | 11 |
| **Total** | **25** | **12** | **13** |

# 3 Findings

## 3.1 High Risk

### 3.1.1 Potential storage collision in new `InfraredBERAWithdrawor` contract

**Severity:** High Risk

**Context:** InfraredBERAWithdraworLite.sol#L84-L86

**Description:** When upgrading from `InfraredBERAWithdraworLite` to `InfraredBERAWithdrawor`, the first new implementation writes over slot 26 (`minActivationBalance`) but leaves slots 27 and 28 untouched. In the old `InfraredBERAWithdraworLite` contract those slots held the `nonceRequest`, `nonceSubmit` and `nonceProcess` counters each initialized to 1. After you call `initializeV2`, slot 26 will be correctly set to the new minimum activation balance, but slots 27 and 28 remain at 1 even though the new contract expects them to be part of its `__gap` (reserved) region.

If a future upgrade ever re-uses those gap slots for real state variables, they will start out with the stale value 1 instead of 0, causing a storage collision and potentially breaking invariants, opening unexpected behavior or corrupting accounting.

**Recommendation:** Update the `InfraredBERAWithdrawor.initializeV2` function to explicitly clear those two slots to zero. For example:

```
  function initializeV2(uint256 _minActivationBalance) external initializer {
      InfraredBERAWithdrawor__AccessControl_init();
      minActivationBalance = _minActivationBalance;
+     // clear legacy nonce values left behind in slots 27 & 28
+     __gap[0] = 0;
+     __gap[1] = 0;
  }
```

This ensures all reserved slots start at zero and prevents any future collision or unintended state.

**Infrared Finance:** Fixed in commit 4a7d997e by resetting the already initialized state variables to 0.

**Cantina Managed:** Verified.

## 3.2 Medium Risk

### 3.2.1 Lack of staleness checks on `nextBlockTimestamp` used in every Beacon-proof verification

**Severity:** Medium Risk

**Context:** BeaconRootsVerify.sol#L241, BeaconRootsVerify.sol#L278, BeaconRootsVerify.sol#L326, BeaconRootsVerify.sol#L353

**Description:** Every call to the `BeaconRootsVerify` library that performs a balance or withdrawal proof with:

```
BeaconRootsVerify.verifyValidatorBalance(
    header,
    balanceMerkleWitness,
    validatorIndex,
    stake,
    balanceLeaf,
    nextBlockTimestamp
);
```

trusts the external caller, in this case the keeper, to choose `nextBlockTimestamp`. Because the contract ultimately checks the supplied root against the EIP-4788 ring buffer, any root whose timestamp is at most `HISTORY_BUFFER_LENGTH = 8191` slots old passes the guard. On Berachain's 2-second slot time this window spans around 4.5 hours. Therefore, the keeper can always pick any root in that window.

If the validator's effective_balance drops inside that window, because it was forced to exit due a higher-priority validator filling the cap, the proof built against the old header still passes `BeaconRootsVerify`. The subsequent call to the withdrawal request precompile succeeds, but when the consensus layer later processes the request it silently discards it as invalid. The execution-layer transaction has already completed, so Infrared's internal accounting decrements stake and issues a

withdrawal ticket. At this point, the accounting between the Execution Layer and the Consensus Layer is broken.

**Proof of concept:**

1. IBera tokens are burnt and therefore multiple withdrawal requests are queued in the `InfraredBERAWithdrawor` contract.
2. Keeper calls `InfraredBERAWithdrawor.execute` to pull liquidity to be able to process those withdrawals and choose a timestamp that is 1 hour old to verify the chosen validator balance (`BeaconRootsVerify.verifyValidatorBalance`). Let's imagine that the amount withdrawn is 100k Bera and belongs to validator 12 (1 hour ago, the actual balance of the validator was 400, 400 - 100 = 300 > 250).
3. However just 1 minute before the `InfraredBERAWithdrawor.execute` call, this validator 12 was forced to exit due to a higher-priority validator joining the active pool and filling the cap.
4. `InfraredBERAWithdrawor.execute` call is succesfull as the call to the precompile does not revert. However, that withdrawal request is ignored at the Consensus Layer as the validator is already exited. As the `InfraredBERAWithdrawor.execute` call did not revert, there is a valid withdrawal ticket that the user can unfairly claim. Moreover, the `InfraredBERAV2` contract already registered the decreased stake of the 100k Bera.

   ```
   IInfraredBERAV2(InfraredBERA).register(validator.pubkey, -int256(amount));
   ```

5. Once the full exited Bera arrives to the `InfraredBERAWithdrawor`, the keeper will call `sweepForcedExit` but only a part of the exited funds will be sent to the `InfraredBERADepositorV2`, as the 100k Bera was already decreased in the `InfraredBERAV2` contract from the `_staked[pubkeyHash]` mapping.

**Recommendation:** Consider introducing an explicit freshness bound for all Beacon proof verifications. For example:

```
uint256 age = block.timestamp - nextBlockTimestamp;
if (age > MAX_ROOT_AGE) revert Errors.StaleBeaconRoot(age);
```

On the other hand, keepers should always choose the validators with the highest stake from the active pool when executing a withdrawal to avoid this scenario.

**Infrared Finance:** Fixed in commit bc92c8cb by implementing the recommended solution. A timestamp older than 10 minutes will not be accepted.

**Cantina Managed:** Verified.

### 3.2.2 `msg.value` is misaccounted as user reserves within `InfraredBERAWithdrawor.execute`

**Severity:** Medium Risk

**Context:** InfraredBERAWithdrawor.sol#L69-L71, InfraredBERAWithdrawor.sol#L247-L250

**Description:** `InfraredBERAWithdrawor.execute` is a payable function. The kepper must attach some `msg.value` (a flat fee) that will later be forwarded to the withdrawal precompile. The function immediately measures the contract's Bera balance via `reserves()` and uses it to assert the relationship between funds on hand and the queued ticket obligations:

```
uint256 queuedAmount = getQueuedAmount();
uint256 _reserves = reserves();           // includes msg.value sent by the keeper to pay the withdrawal
↪  precompile fee
if (queuedAmount < _reserves) {
    revert Errors.ProcessReserves();      // reserve > queue → impossible by design
}

if (amount > (queuedAmount - _reserves + 1 gwei)) {
    revert Errors.InvalidAmount();
}
```

Because `_reserves` already includes `msg.value`, the balance is inflated by the very fee that will be consumed moments later. As the excess fee will be always refunded to the keeper, this could be abused to withdraw from the validators an amount way higher than the needed to back all the pending withdrawal tickets.

**Recommendation:** Exclude the fee from the reserve calculation:

```
uint256 _reserves = reserves() - msg.value;
```

**Infrared Finance:** Fixed in commit e953f9b1 by implementing the recommended solution.

**Cantina Managed:** Verified.

### 3.2.3 State tracking vulnerability in withdrawal processing

**Severity:** Medium Risk

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

**Description:** The `InfraredBERAWithdrawor` contract contains a design issue in the `claim` and `claimBatch` functions where the processing state validation relies on comparing the ticket receiver address against the current depositor address. This approach creates a potential vulnerability when the depositor address changes between the time a withdrawal request is queued and when it is processed.

The current implementation uses a single `PROCESSED` state for all finalized withdrawal requests, then determines whether a ticket should be claimable by checking if the receiver matches the current depositor address. However, if the depositor address is updated after tickets have been queued but before they are processed, tickets originally intended for the old depositor may become claimable by users, or tickets intended for users may become unrecoverable if the old depositor contract is no longer accessible.

This issue is particularly problematic in upgradeable systems where the depositor contract address might change during protocol upgrades. The lack of explicit state tracking for the processing method means that the contract cannot distinguish between tickets that should be claimed by users versus those that should be handled by the depositor for rebalancing purposes.

The same vulnerability exists in the `claimBatch` function, where the address comparison logic could lead to incorrect claim processing if the depositor address has changed since the tickets were originally created.

**Recommendation:** Introduce two distinct processed states:

- `PROCESSED_CLAIM`: for user claims.
- `PROCESSED_DEPOSITOR`: for depositor-driven rebalancing.

Set this state deterministically during request processing and validate it during claims, eliminating reliance on address comparisons that can change over time.

**Infrared Finance:** Fixed in commit b0cf5ae2 by introducing the `CLAIMED` state when `ticket.receiver == depositor`, this effectively prevent calling `claim` for none in `PROCESSED` state requests. The `depositor` check is now removed and will only rely on the state.

**Cantina Managed:** Verified.

### 3.2.4 Full exit blocked by minimum activation balance check

**Severity:** Medium Risk

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

**Description:** The `InfraredBERAWithdrawor.sol` contract contains a logical flaw in the `execute` function where validators with stakes below the minimum activation balance cannot perform full exits. The current implementation applies the minimum activation balance check to all withdrawal amounts, including full exits indicated by `amount == 0`.

When a validator's stake falls below the `minActivationBalance` threshold, any withdrawal attempt will revert with `Errors.WithdrawMustLeaveMoreThanMinActivationBalance()`, even for full exits. This creates a problematic scenario where validators with insufficient stakes cannot exit the system entirely, potentially leaving them in a state where they cannot recover their remaining funds.

The issue is particularly concerning because full exits (indicated by `amount == 0`) represent the complete withdrawal of a validator's stake, making the minimum activation balance requirement irrelevant. The check should not apply to full exits since the validator is exiting completely and will no longer need to maintain the minimum activation balance.

This restriction could prevent validators from exiting when their stakes have been reduced below the minimum threshold due to slashing, penalties, or other consensus layer mechanisms, potentially trapping funds in the system.

**Recommendation:** Add logic to bypass the `minActivationBalance` check when `amount == 0`. This allows validators to exit entirely, regardless of stake size, while preserving the existing guardrails for partial withdrawals.

**Infrared Finance:** Fixed in 3e3afab6 by skipping the check when we do a full withdrawal.

**Cantina Managed:** Verified.

## 3.3 Low Risk

### 3.3.1 Potential griefing and DoS vector in `claim`/`claimBatch` functions

**Severity:** Low Risk

**Context:** InfraredBERAWithdrawor.sol#L372, InfraredBERAWithdrawor.sol#L410

**Description:** `InfraredBERAWithdrawor` exposes two public entry points for withdrawing processed tickets:

```
function claim(uint256 requestId) external whenNotPaused { ... }
function claimBatch(uint256[] calldata requestIds) external whenNotPaused { ... }
```

Both functions perform no authentication check on the `msg.sender` with respect to the ownership of the ticket(s) identified by `requestId` or `requestIds`. This means anyone can invoke, for example, `claim(2)` seconds before another user attempts a more gas-efficient `claimBatch([1,2,3,4])` call. Because `claim` consumes and deletes the ticket record, the subsequent `claimBatch` reverts on ticket 2, forcing the `claimBatch` transaction to revert.

Furthermore, if the designated receiver is a smart contract, its `receive` or `function` could deliberately revert, turning every batch claim into a denial-of-service against that user.

**Recommendation:** Require that `msg.sender` equals the ticket's receiver unless `msg.sender` holds the `KEEPER_ROLE`. A minimal patch looks like:

```
if (!hasRole(KEEPER_ROLE, msg.sender) && msg.sender != tickets[requestId].receiver) {
    revert Errors.UnauthorisedClaimer();
}
```

On the other hand, to mitigate the DoS risk when transferring ETH to a contract receiver, replace any raw transfer or `safeTransferETH` calls with Solady's forceSafeTransferETH, which ensures delivery without executing the recipient's fallback and avoids reverts.

**Infrared Finance:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.3.2 Change in previewBurn return can brick downstream integrations

**Severity:** Low Risk

**Context:** InfraredBERAV2.sol#L330

**Description:** In `InfraredBERAV2` the `previewBurn()` view function was refactored from:

```
// V1
function previewBurn(uint256 shares)
        external
        view
        returns (uint256 beraAmount, uint256 fee);
```

to:

```
// V2
function previewBurn(uint256 shares)
        external
        view
        returns (uint256 beraAmount);
```

The second return value (fee) was removed in the new V2 version. Contracts and off-chain services that were compiled against the V1 interface will still attempt to decode two 32-byte stack slots from the return-data. Because the new implementation only returns one value now, any previous integrator will revert when calling the `previewBurn` function.

**Recommendation:** Ensure that all the integrators are aware of this update and they are upgraded accordingly.

**Infrared Finance:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.3.3 `sweepUnaccountedForFunds` can drain Bera that is already earmarked for outstanding tickets

**Severity:** Low Risk

**Context:** InfraredBERAWithdrawor.sol#L487-L491

**Description:** `InfraredBERAWithdrawor.sweepUnaccountedForFunds` lets the governor transfer "excess" Bera to the protocol's revenue receiver. The guard only verifies that the requested amount does not exceed `reserves()`:

```
if (amount > reserves()) {
    revert Errors.InvalidAmount();
}
```

`reserves()` returns the contract's total Bera balance, which includes idle reserves that genuinely belong to governance and funds that have already been committed to users through withdrawal tickets still sitting in the queue (`getQueuedAmount()`). Nothing prevents the governor from sweeping an amount that is smaller than `reserves()` yet larger than `reserves() getQueuedAmount()`, or which is the same, part of the Bera needed to honour pending withdrawal tickets.

**Recommendation:** Consider treating queued tickets as liabilities and make them ineligible for sweeping. A straightforward fix is:

```
uint256 freeReserves = reserves() - getQueuedAmount();
if (amount > freeReserves) {
    revert Errors.InvalidAmount();
}
```

**Infrared Finance:** Fixed in commit 9ffd4593 by implementing the recommended solution.

**Cantina Managed:** Verified.

### 3.3.4 Merkle tree incomplete root calculation

**Severity:** Low Risk

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

**Description:** The `MerkleTree` library contains a flaw in its root calculation logic when processing datasets with odd numbers of leaves. The current implementation in the `push` function processes only the first count/2 pairs during each level of tree construction, effectively dropping the last leaf when the total number of leaves is odd.

This behavior creates a fundamental inconsistency where the calculated Merkle root corresponds to a different dataset than the one provided. The root represents a tree that is missing the final element, which violates the core principle that a Merkle root should uniquely represent the complete set of input data.

In standard Merkle tree implementations, when a level has an odd number of nodes, the last node is typically duplicated to maintain an even number of nodes at each level, ensuring the tree remains complete and the root accurately represents all input data. The current implementation fails to implement this standard practice, leading to incorrect root calculations.

While this issue may not be directly exploitable in the current codebase due to the specific use case of always passing arrays with length 8 (a power of 2), it represents a significant design flaw that could cause issues if the library is used with datasets of arbitrary sizes in the future.

**Recommendation:** Update the implementation to either:

- Revert when an odd number of leaves is provided (explicit error handling), or...
- Duplicate the last node during tree construction to ensure complete levels.

The former is more robust for security-sensitive contexts, while the latter aligns with standard practices in many Merkle tree implementations.

**Infrared Finance:** Fixed in commit 5d645e8b by reverting when length is odd.

**Cantina Managed:** Verified.

#### 3.3.5 Full exit bypass of withdrawal amount validation

**Severity:** Low Risk

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

**Description:** The `InfraredBERAWithdrawor` contract contains a flaw in the `execute` function where full exit withdrawals (indicated by `amount == 0`) can bypass the withdrawal amount validation logic. The validation check compares the withdrawal amount against the available queued amount minus reserves, but when `amount` is zero, this check becomes ineffective.

The problematic validation occurs in the `execute` function where the code checks if the withdrawal amount exceeds the available queued amount minus reserves plus a 1 gwei tolerance. However, when `amount == 0` (indicating a full exit), the condition `0 > (queuedAmount - _reserves + 1 gwei)` will always be false, allowing the execution to proceed regardless of the actual stake amount being withdrawn.

The issue creates a scenario where keepers could execute full exits even when the contract lacks sufficient reserves to cover the actual stake amount, potentially leading to liquidity issues or incorrect accounting within the withdrawal system.

**Recommendation:** Modify the validation logic to properly handle full exit scenarios by checking the actual stake amount instead of the zero amount parameter. The validation should compare the validator's total stake against the available queued amount minus reserves when `amount == 0`, ensuring that full exits are subject to the same reserve requirements as partial withdrawals.

Alternatively, implement a separate validation path for full exits that explicitly checks if the validator's stake amount can be accommodated within the available reserves, preventing the bypass of the withdrawal amount validation.

**Infrared Finance:** Fixed in commit 035bfb17 by using the full stake amount during the reserves check instead of amount being zero.

**Cantina Managed:** Verified.

### 3.4 Informational

#### 3.4.1 Flat share-denominated burn fee can become ineffective as the exchange rate drifts

**Severity:** Informational

**Context:** InfraredBERAV2.sol#L233-L235

**Description:** `InfraredBERAV2` imposes a burn charge that is hard-coded as an absolute number of shares. Currently the fee is applied as:

```
function burn(address receiver, uint256 shares)
    external
    returns (uint256 nonce, uint256 amount)
{
    if (!withdrawalsEnabled) revert Errors.WithdrawalsNotEnabled();

    // check min exit fee is met in ibera
    uint256 fee = burnFee;
    if (shares < fee) revert Errors.MinExitFeeNotMet();
    uint256 netShares = shares - fee; // <<<
    // ...
}
```

Because the fee is denominated in shares, its economic weight is entirely governed by the protocol's internal exchange rate (`1 share  assets / totalShares`). If the share price appreciates, the fixed fee can become too high, discouraging legitimate exits. Conversely, if the share price depreciates, the fee collapses to negligible value and no longer deters spam-sized `burn()` calls, re-opening the very DoS vector the flat amount was meant to block.

**Recommendation:** Consider monitoring the exchange rate and adjust the flat burn fee accordingly.

**Infrared Finance:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.4.2 Static minActivationDeposit can become stale

**Severity:** Informational

**Context:** InfraredBERADepositorV2.sol#L58

**Description:** `InfraredBERADepositorV2` hard-codes the minimum second-stage stake that must be supplied in `execute()`:

```
// InfraredBERADepositorV2.sol
// needs to be enough to guarentee activation (250k) + inclusion in active set (depends on competition)
minActivationDeposit = 500_000 ether;
// ...
if (stake == InfraredBERAConstants.INITIAL_DEPOSIT) {
    if (amount < minActivationDeposit) {
        revert Errors.DepositMustBeGreaterThanMinActivationBalance();
    }
}
```

The `minActivationDeposit` value is calibrated off-chain under the assumption that 500k Bera comfortably exceeds the lowest stake in the current active validator set. The active set, however, is dynamic: another participant (or even the validator with the lowest stake in the active set) can front-run or simply outbid with a deposit of, say, 510k Bera in the same block. The depositor's `execute()` transaction will still succeed because the contract never re-evaluates the required threshold on-chain, but the resulting validator will fail to enter the active set. Operators would then have to send a third deposit transaction to top-up the validator.

**Recommendation:** Constantly monitor the lowest stake in the current active validator set with a housekeeping script and adjust the `minActivationDeposit` value accordingly.

**Infrared Finance:** Acknowledged. Added as an operational check in github issue 607.

**Cantina Managed:** Acknowledged.

### 3.4.3 Broken accounting between EL and CL if `PENDING_PARTIAL_WITHDRAWALS_LIMIT` is reached

**Severity:** Informational

**Context:** InfraredBERAWithdrawor.sol#L284

**Description:** When `PENDING_PARTIAL_WITHDRAWALS_LIMIT` is reached in the consensus layer, any new partial-withdrawal request submitted via the withdraw precompile is silently ignored, yet the execution-layer transaction still succeeds. In `InfraredBERAWithdrawor.execute`, immediately after calling the precompile, the contract invokes a call to `IInfraredBERAV2(InfraredBERA).register(pubkey, -int256(amount))` which decrements the recorded stake and enqueues a withdrawal ticket.

Because the consensus layer never actually enqueues the withdrawal, no funds are ever released back to the `InfraredBERAWithdrawor` contract. The contract's internal state now believes that stake has been withdrawn, even though on-chain (beacon chain) the validator's balance remains intact. This would break the accounting between the Execution Layer and the Consensus Layer.

**Recommendation:** Consider monitoring the consensus layer and ensure that the `PENDING_PARTIAL_WITHDRAWALS_LIMIT` was not reached or is close to be reached before triggering a partial withdraw through a `InfraredBERAWithdrawor.execute` call.

**Infrared Finance:** Acknowledged. Added as an operational check in github issue 607.

**Cantina Managed:** Acknowledged.

### 3.4.4 `previewBurn` does not respect `withdrawalsEnabled` flag

**Severity:** Informational

**Context:** InfraredBERAV2.sol#L327-L334

**Description:** The `previewBurn(uint256 shareAmount)` function in `InfraredBERAV2` computes how many assets would be returned for a given share amount, but it never checks whether withdrawals are currently enabled. Under EIP-4626, "preview" methods should mirror the conditions under which the corresponding action would succeed or revert. As written:

```
function previewBurn(uint256 shareAmount) public view returns (uint256) {
    uint256 assets = convertToAssets(shareAmount);
    uint256 fee = previewFee(shareAmount);
    return assets > fee ? assets - fee : 0;
}
```

If `withdrawalsEnabled` is `false`, an actual call to `burn` would revert or disallow the operation, yet `previewBurn` will still return a nonzero asset estimate. This mismatch can mislead integrators into believing a `burn` is possible when it will fail at execution time, leading to confusing user experiences or failed transactions.

**Recommendation:** Align `previewBurn` with the contract's withdrawal gating logic by checking `withdrawalsEnabled` at the top of the function. If withdrawals are disabled, it should revert or return zero. For example:

```
  function previewBurn(uint256 shareAmount) public view returns (uint256) {
+     if (!withdrawalsEnabled) {
+         return 0;
+     }
      uint256 assets = convertToAssets(shareAmount);
      uint256 fee = previewFee(shareAmount);
      return assets > fee ? assets - fee : 0;
  }
```

**Infrared Finance:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.4.5 Missing zero-address check for receiver in `InfraredBERAV2.burn` function

**Severity:** Informational

**Context:** InfraredBERAV2.sol#L226

**Description:** The `InfraredBERAV2.burn` function does not guard against `receiver == address(0)`. While currently a user would normally pass their own address, allowing `address(0)` opens a future problem: if `burn(address(0), ...)` were ever used in conjunction with the `InfraredBERAWithdrawor.claimBatch` flow, claims intended for the zero address could effectively be "stolen" by any caller.

**Recommendation:** Insert an explicit check at the start of burn to reject the zero address:

```
  function burn(address receiver, uint256 shares) external returns (uint256) {
+     if (receiver == address(0)) {
+         revert Errors.InvalidReceiver();
+     }
      // ...
  }
```

**Infrared Finance:** Fixed in commit 147fcd1d by implementing the recommended solution.

**Cantina Managed:** Verified.

### 3.4.6 No cap on dynamic withdrawal-request fee in execute

**Severity:** Informational

**Context:** InfraredBERAWithdrawor.sol#L261-L263

**Description:** In the `InfraredBERAWithdrawor` contract the `execute` function is marked as `payable` so that it can forward whatever Bera was sent as the dynamic fee to the EIP-7002 withdrawal precompile. However,

there is no guard against an abnormally large withdrawal fee. Under heavy-use or deliberate griefing, the fee formula in the precompile can spike exponentially.

**Recommendation:** Introduce an explicit upper bound on the fee the contract will accept and forward. For example, define a sane maximum in the contract (e.g. `uint256 constant MAX_WITHDRAWAL_FEE = 1 ether;`) and then in execute before calling the precompile:

```
  function execute(/*...*/) external payable onlyKeeper whenNotPaused {
+     uint256 feePayable = getFee();
+     require(feePayable <= MAX_WITHDRAWAL_FEE, Errors.FeeTooHigh());
      // existing stake- and proof-validation logic...
      WITHDRAW_PRECOMPILE.call{ value: feePayable }(/*...*/);
      // ...
  }
```

**Infrared Finance:** Acknowledged. Added as an operational check in github issue 607.

**Cantina Managed:** Acknowledged.

### 3.4.7 Unused imports

**Severity:** Informational

**Context:** InfraredBERADepositorV2.sol#L4, InfraredBERADepositorV2.sol#L8

**Description:** The `InfraredBERADepositorV2` imports modules that aren't referenced anywhere in the contract:

```
import {IInfraredBERADepositor} from "src/interfaces/IInfraredBERADepositor.sol";
import {SafeTransferLib} from "@solmate/utils/SafeTransferLib.sol";
```

**Recommendation:** Remove both import statements.

**Infrared Finance:** Fixed in commit 9b49d9b9 by implementing the recommended solution.

**Cantina Managed:** Verified.

### 3.4.8 Missing upper-bound validation on `minActivationDeposit` setter

**Severity:** Informational

**Context:** InfraredBERADepositorV2.sol#L285-L291

**Description:** In `InfraredBERADepositorV2.setMinActivationDeposit` function, the governor can set `minActivationDeposit` to any value:

```
minActivationDeposit = _minActivationDeposit;
```

However, elsewhere the contract enforces that a second deposit plus the existing stake must not exceed `MAX_EFFECTIVE_BALANCE` (10.000.000 Bera):

```
// The validator balance + amount must not surpass MaxEffectiveBalance of 10 million BERA.
if (stake + amount > InfraredBERAConstants.MAX_EFFECTIVE_BALANCE) {
    revert Errors.ExceedsMaxEffectiveBalance();
}
```

If the governor sets `minActivationDeposit` higher than `MAX_EFFECTIVE_BALANCE - INITIAL_DEPOSIT`, then any call to execute for a fresh validator (with `stake == INITIAL_DEPOSIT`) will always revert.

**Recommendation:** Add a require check in the `initializev2` and in the setter function to ensure that `_minActivationDeposit` cannot exceed the available headroom:

```
- minActivationDeposit = _minActivationDeposit;
+ require(
+     _minActivationDeposit <= InfraredBERAConstants.MAX_EFFECTIVE_BALANCE
+                            - InfraredBERAConstants.INITIAL_DEPOSIT,
+     "minActivationDeposit: exceeds max effective balance"
+ );
+ minActivationDeposit = _minActivationDeposit;
```

**Infrared Finance:** Fixed in commit 3225b478 by implementing the recommended solution.

**Cantina Managed:** Verified.

### 3.4.9 Unnecessary request ID check in accumulated amount calculation

**Severity:** Informational

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

**Description:** The `InfraredBERAWithdrawor` contract contains an unnecessary conditional check in the `queue` function when calculating the accumulated amount for withdrawal requests. The current implementation uses a ternary operator to handle the special case when `requestId == 1`, but this check is redundant due to the default behavior of Solidity's mapping access.

When `requestId` is 1, accessing `requests[0].accumulatedAmount` will return the default value of 0 for the `uint128` type, since no request with ID 0 has been stored in the mapping. This means the calculation `requests[0].accumulatedAmount + amount` will correctly result in `0 + amount = amount`, which is exactly what the current conditional logic achieves.

**Recommendation:** Remove the conditional check and simplify the accumulated amount calculation to directly use `requests[requestId - 1].accumulatedAmount + amount`. This approach leverages Solidity's built-in behavior where accessing a non-existent mapping key returns the default value, eliminating the need for explicit boundary condition handling.

**Infrared Finance:** The check was removed in commit 145bfb55.

**Cantina Managed:** Verified.

### 3.4.10 Redundant state check in withdrawal processing

**Severity:** Informational

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

**Description:** The `InfraredBERAWithdrawor.sol` contract contains an unnecessary state validation check in the `process` function. The code verifies that each request is in `QUEUED` state before processing it, but this check appears redundant given the contract's request management system.

The contract uses a sequential request ID system where `requestsFinalisedUntil` tracks the highest processed request ID. All requests from `requestsFinalisedUntil + 1` to `requestLength` should logically be in `QUEUED` state, as the `process` function is the only mechanism that transitions requests from `QUEUED` to `PROCESSED` state. The `requestsFinalisedUntil` variable ensures that requests are processed in order and prevents double-processing.

The current implementation also performs a balance check using the total delta amount before processing individual requests. If the state check is necessary due to potential edge cases where requests might not be in `QUEUED` state, then the balance validation should also account for only the `QUEUED` requests rather than the total delta between the finalised indices.

The state check adds unnecessary gas overhead and complexity without providing clear functional benefits, as the request management system should maintain invariant states based on the `requestsFinalisedUntil` tracking mechanism.

**Recommendation:** Remove the redundant state check unless there is a specific justification for requests potentially being in non-`QUEUED` states within the valid processing range. If the check is removed, the balance validation can remain as is since all requests in the processing range should be in `QUEUED` state.

If the state check is necessary due to edge cases not apparent in the current codebase, consider adding documentation explaining why requests might not be in `QUEUED` state and adjust the balance validation to only consider `QUEUED` requests when calculating the required reserves.

**Infrared Finance:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.4.11 `InfraredBERAWithdrawor.execute` can unintentionally trigger an immediate forced exit

**Severity:** Informational

**Context:** InfraredBERAWithdrawor.sol#L189

**Description:** `InfraredBERAWithdrawor.execute` permits a keeper to withdraw any amount provided that the post-withdrawal balance `stays minActivationBalance` (250k Bera):

```
if (stake - amount < minActivationBalance) {
    revert Errors.WithdrawMustLeaveMoreThanMinActivationBalance();
}
```

Yet Berachain enforces a hard validator-set cap of 69 entries. At the end of every epoch `processValidatorSetCap` sorts the projected next-epoch set by `effective_balance` and calls `InitiateValidatorExit` on the lowest-stake validators until the cap is met, see state_processor_validators.go

If a partial withdrawal leaves a validator only slightly above `minActivationBalance` it may still be the smallest stake in the set. As soon as a new validator with `minActivationBalance` tries to join, the sorter will place the freshly topped-up entrant ahead of the depleted validator. Then the cap logic will force-exit the latter in the next epoch even though it met the contract's `minActivationBalance`.

**Recommendation:** When calling the `InfraredBERAWithdrawor.execute`function, ensure that the validator is never left with the lowest `effective_balance` of the active set. Consider leaving that validator with a safety buffer so that it is not exited in the short term.

**Infrared Finance:** Acknowledged. Added as an operational check in github issue 607.

**Cantina Managed:** Acknowledged.

### 3.4.12 Inefficient withdrawal processing

**Severity:** Informational

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

**Description:** The current implementation of the `InfraredBERADepositorV2` and `InfraredBERAWithdrawor` contracts does not allow rebalancing Bera from the depositor queue to the withdrawor. This restriction compels the system to handle withdrawal requests by executing multiple partial withdrawals from active validators, with each withdrawal incurring a withdrawal fee. This approach is both inefficient and expensive, particularly when sufficient funds are available in the depositor queue to directly fulfill withdrawal requests. For instance, imagine a situation where 10000 Bera are queued for withdrawal and the depositor queue holds 12000 Bera. Rather than transferring 10000 Bera directly from the depositor queue to the withdrawor, the system must for example process 20 separate partial withdrawals of 500 Bera each from active validators, each carrying its own fee. This unnecessarily increases costs and complicates the process.

Implementing the ability to transfer Bera from the depositor queue to the withdrawor offers several advantages. First, it reduces fees by eliminating the need for multiple partial withdrawal transactions from active validators. Second, it improves efficiency by simplifying the withdrawal process, allowing requests to be satisfied directly from available queued funds. Third, it enhances validator stability by reducing the frequency and volume of withdrawals from active validators, which could otherwise lead to validator exits if large partial withdrawals are frequent.

**Recommendation:** To resolve this issue, the `InfraredBERADepositorV2` contract should be updated to enable Bera transfers from the depositor queue to the withdrawor. This modification would require adding a specific function, only callable by the keepers, that:

1. Decreases `InfraredBERADepositorV2` reserves.
2. Transfers the respective amount of Bera to the `InfraredBERAWithdrawor` contract.

**Infrared Finance:** Acknowledged. Yes, pulling from depositor queue for withdrawal tickets would be more efficient. We had previously considered this in PR 584. It was our design choice to keep deposit and withdraw channels separate for simplicity in security, accounting and operations at the expense of some efficiency. We might consider to add this feature in the future.

**Cantina Managed:** Acknowledged.

### 3.4.13 EIP 7002 withdrawals are rate-limited by the consensus constant `MaxPendingPartialsPerWithdrawalsSwee`

**Severity:** Informational

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

**Description:** Every partial-withdrawal requested through the EIP-7002 precompile(`WITHDRAW_PRECOMPILE`) is first accepted by `InfraredBERAWithdrawor.execute()` and immediately deducted from Infrared's internal stake accounting:

```
// InfraredBERAWithdrawor.execute
IInfraredBERAV2(InfraredBERA).register(pubkey, -int256(amount));
```

Once the on-chain call succeeds the request lives in the beacon state's `pendingPartialWithdrawals[]` queue, it is not yet a real withdrawal and no Bera has been credited to the Withdrawor's balance.

The consensus engine subsequently materialises at most `MaxPendingPartialsPerWithdrawalsSweep` entries from that queue in each block. The throttling point is within `beacon-kit-1.2.0/state-transition/core/state/statedb.go - consumePendingPartialWithdrawals` function:

```
for _, withdrawal := range ppWithdrawals {
    if withdrawal.WithdrawableEpoch > epoch ||
       len(withdrawals) == constants.MaxPendingPartialsPerWithdrawalsSweep {
        break            // ← hard stop after N items
    }
    ... append to block's Withdrawal list ...
}
```

`MaxPendingPartialsPerWithdrawalsSweep` is a chain constant defined in `primitives/constants` (Berachain main-net value = 8):

```
// Withdrawals processing:
// https://github.com/ethereum/consensus-specs/blob/dev/specs/electra/beacon-chain.md#withdrawals-processing
const (
    // MaxPendingPartialsPerWithdrawalsSweep is the maximum number of pending partial withdrawals
    // per sweep.
    MaxPendingPartialsPerWithdrawalsSweep = 8
)
```

Because it is evaluated once per block, a backlog of `k` partial requests will take `ceil(k / 10)` blocks before the corresponding Bera is forwarded to the `InfraredBERAWithdrawor` contract. While the protocol remains correct, the funds will eventually arrive, this rate-limit has some operational side-effects:

1. `InfraredBERAWithdrawor.process()` can only finalize user tickets when `address(this).balance - totalClaimable` is large enough. Large bursts of exits therefore sit queued for multiple blocks even though sufficient liquidity already exists on the consensus layer.
2. An adversary able to spam partial withdrawals (e.g. by splitting a full exit into > 256 partials) can deterministically depress Withdrawor liquidity for > 25 blocks, creating a temporary DoS window on user redemptions.

**Recommendation:** Mitigate the throughput bottleneck rather than trying to bypass the consensus rule as it is part of the fork logic and cannot be disabled. Three complementary measures are suggested:

1. Introduce a Depositor → Withdrawor fast-path: Provide a `rebalanceToWithdrawor(uint256 amount)` function in InfraredBERADepositorV2 callable by a keeper/governor. Moving idle `reserves` directly into the Withdrawor allows `process()` to settle tickets immediately, side-stepping the consensus throttle and avoiding many precompile calls.
2. Integrate queue-depth awareness into keeper logic: Enhance the off-chain keeper to monitor `pendingPartialWithdrawals.length`, either via light-client proof or RPC, and dynamically adjust its behavior based on current queue saturation. The keeper's algorithm responsible for selecting the validator and amount for withdrawal precompile calls should incorporate current queue depth as a factor in its optimization strategy.

**Infrared Finance:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.4.14 Staking is rate-limited by the consensus constant `MaxDepositsPerBlock`

**Severity:** Informational

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

**Description:** Every time a user mints IBera, the Bera is pushed into `InfraredBERADepositor.queue{value:...}()`. The amount is then forwarded to the Berachain `Deposit` contract by a keeper via `InfraredBERADepositor.execute` call.

Once that Execution Layer transaction is executed the event becomes part of the deposit log and must be "ingested" by the beacon chain. That ingestion is performed, block-by-block, inside the state-transition function `processOperations`. The consensus rules enforce a strict upper bound on how many `Deposit` objects can appear in a single beacon block:

```
// state-transition/core/state_processor_staking.go
deposits := blk.GetBody().GetDeposits()
if uint64(len(deposits)) > sp.cs.MaxDepositsPerBlock() {
    return errors.Wrapf(
        ErrExceedsBlockDepositLimit,
        "expected  %d, got %d",
        sp.cs.MaxDepositsPerBlock(), len(deposits),
    )
}
```

`MaxDepositsPerBlock` is defined in the chain specification (e.g. `16` on the current Berachain network). When more than 16 new deposit events exist in the log, the proposer is forced to carry only the first 16, the remainder must wait for subsequent blocks. Unlike the withdrawal path, there is no bounded in-state queue that can overflow or drop entries, excess deposits simply accumulate back-pressure until cleared at a constant rate of `16` per block.

The implication for Infrared is that during periods of very high inflow `InfraredBERADepositor.reserves` could remain positive for many blocks. While the contractual promise "one IBera = one Bera staked" is ultimately preserved, an operational effect emerge: Newly minted IBera begins accruing staking yield only after the corresponding deposit is confirmed on the beacon chain. A protracted log backlog therefore reduces APY for all Ibera holders. The `deposits` field inside `InfraredBERAV2` continues to rise immediately, because `_deposit()` is executed at mint time. Until the beacon state catches up, `convertToAssets()` and related accounting over-estimate the on-chain validator balance, albeit temporarily.



**Recommendation:** Merely an informational issue. A possible mitigation against this would be to update `InfraredBERADepositorV2` with a function allowing idle `reserves` to be shifted to the Withdrawor contract (`rebalanceToWithdrawor`).

**Infrared Finance:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.4.15 Minting below flat burn fee will block a future withdrawal

**Severity:** Informational

**Context:** InfraredBERAV2.sol#L232-L235

**Description:** The `InfraredBERAV2` contract applies a fixed "burn fee" in iBERA shares whenever a user calls `burn`:

```
uint256 fee = burnFee;            // flat fee in shares
uint256 netShares = shares - fee; // underflows or reverts if shares  fee
```

However, there is no corresponding minimum enforced during `mint`. This means a user can `mint` an amount of BERA that converts to fewer iBERA shares than the flat `burnFee`. Such users will then be unable to ever exit (unless they purchase more iBERA), because any subsequent call to `burn(shares)` will revert (or underflow), locking their entire position.

In practice, a user who mints e.g. 1 iBERA when the flat burn fee is 5 iBERA will be "stuck": they cannot redeem those shares, and their funds are irrecoverable.

**Recommendation:** Prevent this dead-end scenario by enforcing a minimum mintable/shareable amount equal to the burn fee. For example, in your `mint` implementation:

```
  function mint(uint256 assets, address receiver) external returns (uint256 shares) {
      shares = convertToShares(assets);
+     if (shares <= burnFee) {
+         revert Errors.AmountTooSmallForBurnFee();
+     }
      _mint(receiver, shares);
      emit Mint(msg.sender, receiver, assets, shares);
  }
```

Similarly, update `previewMint` to return zero (or revert) for any asset amount that would yield `burnFee` shares, so integrators and UIs can prevent users from creating non-exitable positions.

**Infrared Finance:** Acknowledged. Our front-end integration will direct all low burns to swaps instead, which works fine as long as liquidity remains. Should liquidity become too low (e.g. if iBERA is wound down) we can drop the burn fee to close to zero. If we add a minimum mint, it will not apply to shares already minted.

**Cantina Managed:** Acknowledged.