Infrared PR 657

Security Review

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

March 18, 2026

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 Mar 4th to Mar 15th the Cantina team conducted a review of infrared-contracts on commit hash 2504e725. The team identified a total of **31** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	13	13	0
Low Risk	10	6	4
Gas Optimizations	0	0	0
Informational	8	6	2
Total	**31**	**25**	**6**

2.1 Scope

The security review had the following components in scope for infrared-contracts on commit hash 2504e725:

```
src/periphery
├── CuttingBoardNFT.sol
├── CuttingBoardSyndicate.sol
└── libraries
    └── CuttingBoardSyndicateLib.sol
```

3 Findings

3.1 Medium Risk

3.1.1 Live claim below raised minimum bricks start

Severity: Medium Risk

Context: CuttingBoardDutchAuctionV1_1.sol#L99-L107, CuttingBoardDutchAuction.sol#L483-L487, CuttingBoardDutchAuction.sol#L809-L824

Description: `setMinimumPrice` and `claimCuttingBoardControl` do not agree on which price should remain authoritative after the keeper raises the auction floor during a live auction. The keeper path raises both `minimumPrice` and, when needed, `lastClosingPrice` so future auctions cannot start below the new floor. That setter does not reprice an already-live auction, though. The live auction keeps using the `startingPrice` and `basePrice` snapshot that was stored when it was opened. Later, if that live auction is claimed at a lower decayed price, the claim path blindly overwrites `lastClosingPrice` with that lower execution price. The next `startCuttingBoardAuction` call then mixes the stale low `lastClosingPrice` with the newer higher `minimumPrice`, and the auction cannot be created because the computed starting price is no longer strictly above the base price.

```
function setMinimumPrice(uint256 _minimumPrice) external virtual onlyKeeper {
    if (_minimumPrice == 0) revert InvalidMinimumPrice();
    $.minimumPrice = _minimumPrice;
    if (_minimumPrice > $.lastClosingPrice) {
        $.lastClosingPrice = _minimumPrice;
    }
}
```

```
auction.claimed = true;
auction.claimPrice = uint128(currentPrice);
$.lastClosingPrice = currentPrice;
```

```
uint256 priceRange = auction.startingPrice - auction.basePrice;
uint256 priceDrop = (priceRange * elapsed) / $.auctionDuration;
return auction.startingPrice - priceDrop;
```

```
uint256 starting = ($.lastClosingPrice * $.startingPriceMultiplier) / 1e18;
uint256 base = ($.lastClosingPrice * 1e18) / $.basePriceDivisor;

if (base < $.minimumPrice) {
    base = $.minimumPrice;
}

if (starting <= base) revert InvalidPriceRange();
```

This creates a practical liveness failure. Suppose the current auction started when the reference price was `1000`, with a 2x starting multiplier and a 2x base divisor, so the auction opened at `2000` and stores a `basePrice` of `500`. During that live auction, the keeper raises `minimumPrice` to `1001`. Because `1001 > 1000`, the setter also lifts `lastClosingPrice` to `1001`. The live auction still decays along its original curve toward its stored base of `500`, because changing `minimumPrice` does not modify the active auction snapshot. An outsider can then wait until the live price reaches `500` and claim it. The claim succeeds, but it also writes `lastClosingPrice = 500`. When the keeper later tries to start the next auction for any validator, the protocol computes `starting = 1000` and `base = max(250, 1001) = 1001`, so `starting <= base` and the call reverts with `InvalidPriceRange`.

The important point is that the attacker does not need any privileged role. They only need a normal auction claim after governance or the keeper has raised the floor during a live auction. Once that claim lands, global auction progression can halt until the keeper notices the broken state and manually repairs the reference price or lowers the minimum again.

The root cause is that the contract treats `currentPrice` as the universal next-auction reference even when the keeper has already declared that the protocol floor has moved higher. `CuttingBoardDutchAuctionV1_1` only clamps the expired-unclaimed branch inside `startCuttingBoardAuction`. It does not override the live claim path, so a successful live claim can

still push `lastClosingPrice` back below the raised floor and recreate the same `InvalidPriceRange` condition on the next auction start.

Recommendation: Keep the next-auction reference consistent with the active floor. The simplest fix is to ensure the claim path never lowers `lastClosingPrice` below `minimumPrice`, for example by storing `max(currentPrice, minimumPrice)` instead of raw `currentPrice`. As defense in depth, the same invariant can also be enforced at the beginning of `startCuttingBoardAuction` before the next price range is computed. After that change, keep a regression test for the exact sequence where the keeper raises `minimumPrice` during a live auction and a later live claim clears below the new floor.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.1.2 Multiplier live claim can brick next start

Severity: Medium Risk

Context: CuttingBoardDutchAuction.sol#L860-L870

Description: `setStartingPriceMultiplier` accepts values that may look safe when the keeper sets them, but the protocol does not preserve the same reference price afterward. A later live claim can rewrite `lastClosingPrice` to a much higher execution price, and the next `startCuttingBoardAuction` call reuses that new reference with the previously stored multiplier. If the combination is now too large, the next auction start reverts with `PriceOverflow` and auction progression stalls until the keeper manually repairs the configuration.

```
function setStartingPriceMultiplier(uint256 _startingPriceMultiplier)
    external
    virtual
    onlyKeeper
{
    if (_startingPriceMultiplier <= 1e18) revert InvalidMultiplier();
    $.startingPriceMultiplier = _startingPriceMultiplier;
}
```

```
auction.winner = msg.sender;
auction.claimed = true;
auction.claimPrice = uint128(currentPrice);
$.lastClosingPrice = currentPrice;
```

```
if ($.lastClosingPrice > type(uint256).max / $.startingPriceMultiplier) {
    revert PriceOverflow();
}
uint256 starting = ($.lastClosingPrice * $.startingPriceMultiplier) / 1e18;
```

The root cause is that `setStartingPriceMultiplier` validates only the immediate lower bound of the multiplier and assumes the current price reference remains representative. That assumption is false. The claim path later promotes `currentPrice` into the global `lastClosingPrice`, so the setter is really configuring a multiplier against a moving future reference that can become much larger than the price the keeper had in mind.

A realistic failure sequence is straightforward. Suppose the keeper sets a very aggressive multiplier while `lastClosingPrice` is still small, for example because the previous auction cleared at a low value. The next auction starts successfully because the current reference is still low enough. A user then claims that auction very early, while the live price is still near the auction's high starting price. The claim succeeds and stores that high execution price into `lastClosingPrice`. When the keeper later tries to start the next auction, the contract multiplies the new higher `lastClosingPrice` by the previously stored aggressive multiplier and now exceeds the arithmetic or storage bounds enforced during auction creation. The revert happens only at the later start, so the bad configuration can sit latent in state until auction progression is needed again.

This is primarily a privileged configuration ”mistake”, but the resulting failure is externally triggerable once the keeper has set the multiplier. Any ordinary bidder who claims the auction early enough can be the transaction that pushes `lastClosingPrice` into the unsafe range for the next auction start.

Recommendation: Validate `setStartingPriceMultiplier` against realistic future reference prices instead of only checking that the multiplier is greater than `1e18`. At minimum, bound the multiplier so that auction-start calculations remain safe for the largest `lastClosingPrice` that can plausibly be written by a live claim. Another acceptable fix is to normalize the post-claim reference before writing `lastClosingPrice`, so a live claim cannot move the next-auction input into an unsafe range.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.1.3 Active rounds can brick on chef policy drift

Severity: Medium Risk

Context: CuttingBoardManager.sol#L598-L616

Description: `triggerClaim` decides the winning allocation for the round and moves that round into `Active`. At that point, the economic outcome is already settled: the winning partners are chosen, each winner's `allocatedWeight` is fixed and the buffer vault for that round is snapshotted. Later, `submitProposal` rebuilds the proposal from that stored state and sends it to `CuttingBoardManager`.

The problem is that `CuttingBoardManager` does not validate the proposal against the rules that were in place when the round became `Active`. It validates against the current live `BeraChef` policy instead. That means a round can be valid when it is claimed, then become invalid before it is submitted, even though nobody did anything wrong and the round's stored allocation never changed.

Not every policy drift case is equally bad once the round is active. If a winning partner's vault is later removed from the whitelist, the holder of that partner's `SlotNFT` can still call `updateSlotVault` and redirect the share to a fresh whitelisted vault before `submitProposal`. That is operationally inconvenient, but it is recoverable.

The harder cases are the ones the protocol can no longer rewrite after `triggerClaim`. If `BeraChef` lowers `maxNumWeightsPerRewardAllocation`, the frozen winner set may now have too many entries. If `BeraChef` lowers `maxWeightPerVault`, one of the frozen `allocatedWeight` values may now be too large. If the snapshotted buffer vault is later removed from the whitelist while the round still has a non-zero buffer allocation, that receiver is frozen into the round and there is no path to swap it out. In those cases, `submitProposal` reverts even though the round was valid when claimed, and the round remains unsubmitable unless governance abandons it through an administrative escape hatch or waits for the control period to expire.

For example, imagine a round that is claimed while `BeraChef` allows 5 receivers in one allocation. The winning result uses 4 partner vaults plus the buffer vault, so the round is valid and moves to `Active`. Before anyone calls `submitProposal`, governance reduces the live limit to 4 receivers. When `submitProposal` later rebuilds the exact same 5-entry allocation, `CuttingBoardManager` rejects it with `TooManyWeights()`. The round already settled under the old rules, but it can no longer be submitted under the new ones, so the validator's update path is bricked for that round.

Recommendation: Once a round becomes `Active`, its submission path should not depend on mutable `BeraChef` rules that the protocol can no longer satisfy post-claim. At minimum, the constraints tied to frozen state should be snapshotted for that round: receiver-count limits, per-vault weight limits and whitelist validity for the snapshotted buffer receiver. If that is not desirable, the protocol should add an explicit recovery path for active rounds, such as governance-controlled re-composition, buffer-receiver migration, cancellation or refund. Partner-vault whitelist drift can already be handled operationally through `updateSlotVault`, so it should be treated separately from the unrecoverable active-round cases.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.1.4 Open rounds can brick on chef policy drift before claim

Severity: Medium Risk

Context: CuttingBoardSyndicate.sol#L782-L797

Description: `triggerClaim` enforces local fill, entry-count and buffer conditions, but it does not explicitly mirror all live manager-side weight constraints before making the external claim path. This leaves a drift

window where open-round registrations remain locally admissible yet fail later during manager validation when claim is executed.

```
if (fill.bfWeight > maxPerVault) revert BufferWeightExceedsMax(...);
if (fill.count == 0) revert NoBidders();
uint256 totalEntries = fill.count + (fill.bfWeight > 0 ? 1 : 0);
if (totalEntries > $.chef.maxNumWeightsPerRewardAllocation()) revert TooManyPartners();
if (fill.bufferRequired > $.bufferDeposit) revert InsufficientBuffer(...);
```

```
if (weights.length > $.chef.maxNumWeightsPerRewardAllocation()) revert TooManyWeights();
if (weight.percentageNumerator == 0 || weight.percentageNumerator >
↪  $.chef.maxWeightPerVault()) revert InvalidWeight();
if (!$.chef.isWhitelistedVault(weight.receiver)) revert VaultNotWhitelisted();
```

When live whitelist or cap policy changes after slot registration but before claim, rounds can become unclaimable until participants reconfigure or the auction expires. This creates a cross-contract liveness failure that is difficult for operators to predict from local syndicate state.

Recommendation: Pre-validate the exact outgoing claim weight set against the same live predicates manager uses before initiating external claim flow. Keep `triggerClaim`, `canTrigger` and `effectiveTriggerPrice` aligned to one shared feasibility routine so operational readiness signals remain truthful.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.1.5 Live `BeraChef` drift can silently revoke boards

Severity: Medium Risk

Context: CuttingBoardSyndicate.sol#L1018-L1019

Description: The syndicate flow treats a winning board as if it stays live until someone replaces it. `BeraChef` does not. It checks validity at read time and if the active allocation no longer satisfies the current rules, it silently falls back to the default allocation.

```
function getActiveRewardAllocation(bytes calldata valPubkey) external view returns
↪  (RewardAllocation memory) {
    RewardAllocation memory ara = activeRewardAllocations[valPubkey];

    if (ara.startBlock > 0 && _checkIfStillValid(ara.weights)) {
        return ara;
    }

    return defaultRewardAllocation;
}
```

The periphery only checks validity when the proposal is submitted:

```
if (weights.length > $.chef.maxNumWeightsPerRewardAllocation()) revert TooManyWeights();
if (weight.percentageNumerator == 0 || weight.percentageNumerator >
↪  $.chef.maxWeightPerVault()) {
    revert InvalidWeight();
}
if (!$.chef.isWhitelistedVault(weight.receiver)) revert VaultNotWhitelisted();
```

Because of this a board can be valid when proposed, approved and activated, then later become invalid if governance de-whitelists a vault, lowers `maxWeightPerVault`, or lowers `maxNumWeightsPerRewardAllocation`. When that happens, the syndicate round still shows `Active` but `BeraChef` starts returning the default allocation instead of the board the syndicate won.

In practice, users can keep holding what looks like live control over a validator while rewards are already being routed somewhere else. Nothing in the syndicate state machine fails, and there is no obvious onchain signal unless monitoring compares the stored board against the effective allocation returned by `BeraChef`.

Recommendation: Expose whether an active round's stored board is still valid under the current `BeraChef` rules and whether the validator's effective allocation still matches it. Monitoring should alert when an `Active` round has already fallen back to the default allocation.

If this fallback is expected behavior, document it clearly so `Active` is not treated as proof that the won board is still in effect. If stronger guarantees are needed, add a recovery path so the protocol can react before an active round drifts onto the default board.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.1.6 Permissionless vault squatting blocks partners

Severity: Medium Risk

Context: CuttingBoardSyndicate.sol#L581-L589

Description: `registerSlot` enforces that each vault can appear only once per round, but it does not ask whether the caller actually controls that vault or has any authority to use it. As written, any user can reserve any whitelisted vault address as long as no one else has used it yet.

```
if (!$.chef.isWhitelistedVault(vault)) revert VaultNotWhitelisted();
//...
for (uint256 i = 0; i < _partners.length; i++) {
    if ($.slots[auctionId][_partners[i]].vault == vault) {
        revert DuplicateVaultEntry(vault);
    }
}
```

That opens the door to straightforward vault squatting. A user does not need to control the vault they register; they only need to be first. Once the slot is in place, there is no built-in way to evict it during the open round other than waiting for the squatter to leave voluntarily or for the round to expire.

Imagine that Alice operates a well-known whitelisted vault and intends to join a round using that vault. Bob sees the round open first and immediately registers a slot pointing to Alice's vault, even though he has no relationship to it. Alice then tries to register and is rejected because the vault is already taken. Bob does not need to win the auction for this to be harmful. He can simply sit on the slot long enough to block Alice from participating, force her onto a worse vault, or keep her out of the round entirely unless she waits for him to exit or for the round to expire.

The impact is mostly liveness and quality-of-allocation degradation, but it is very usable in practice because the attack is cheap and does not depend on breaking any permission checks. The contract currently treats ”first caller” as equivalent to ”authorized user” for vault selection.

Recommendation: Require some form of vault authorization during registration and open-state vault updates, such as a controller signature or an on-chain ownership check. If fully permissionless registration is a hard design requirement, the protocol should at least add anti-squatting protections such as challenge windows, meaningful reservation costs, or a narrow eviction mechanism for clearly unauthorized occupancy.

Another path is allowing same vaults in bids but only include top 1 in selection or replacing the same vault bids with higher bids but do note that this will require a major code refactor.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.1.7 Tie-break order drift after partner exits

Severity: Medium Risk

Context: CuttingBoardSyndicate.sol#L1176-L1178

Description: The fill logic uses a stable sort, so when two partners have the same bid value the winner is determined by their current order in `roundPartners`. The problem is that this order is not actually stable over the life of the round. It can change whenever some other partner exits, because exits are handled with swap-pop removal.

```
while (j >= 0 && bidValues[uint256(j)] < keyBid) {
//...
}
```

```
if (_partners[i] == partner) {
    _partners[i] = _partners[len - 1];
    _partners.pop();
    return;
}
```

That means the documented "registration order breaks ties" rule is only true until the participant set changes. After that, tied bids can resolve differently even if neither tied bidder touched their slot.

A good example is Alice and Bob each bidding with the same effective value, while Carol is also in the round. Alice registered before Bob, so under the documented behavior Alice should win the tie. If Carol later exits and her slot is removed with swap-pop, Bob can be moved ahead of Alice in `roundPartners` without changing either of their bids. When the round is eventually filled at a price where only one of the tied bids can be fully satisfied, Bob can now be filled ahead of Alice purely because Carol left at the right time. In other words, a participant can change the ordering of tied competitors indirectly, without touching the tied bids themselves.

Under oversubscribed tied bids, the changed ordering can affect who receives the last full fill, who gets partially filled, and who is excluded.

Recommendation: Record an immutable tie-break key when the slot is created, such as a monotonically increasing registration index, and sort by that value after comparing bid value. That keeps tie behavior stable even if partners exit, re-enter, or are removed from the backing array.

Another option would be simply removing the `exitSlot()` function.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.1.8 `SlotNFT` rotation can break active update rights

Severity: Medium Risk

Context: CuttingBoardSyndicate.sol#L456-L459

Description: Governance can change the global `slotNFT` pointer at any time, and active update authorization resolves ownership through the current pointer instead of a per-round snapshot. This means active rounds can lose continuity if the pointer rotates away from the contract that minted their tokens.

```
function setSlotNFT(address _slotNFT) external virtual onlyGovernor {
    _getStorage().slotNFT = CuttingBoardSlotNFT(_slotNFT);
}
```

```
CuttingBoardSlotNFT _slotNFT = $.slotNFT;
uint256 tokenId = _slotNFT.getTokenId(auctionId, originalPartner);
if (tokenId == 0 || _slotNFT.ownerOf(tokenId) != msg.sender) {
    revert NotSlotNFTHolder();
}
```

After rotation, rightful holders of old-round slot NFTs can fail authorization because lookup and ownership checks are executed against the new contract domain. The impact is active-rights disruption and round liveness failure.

Recommendation: Consider taking a snapshot of `slotNFT` per round at trigger time and use the snapshot for all active-round rights checks. Moreover, consider also adding guardrails that block unsafe rotation while active rounds exist.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.1.9 Active round partner bloat degrades liveness

Severity: Medium Risk

Context: CuttingBoardSyndicate.sol#L831-L842

Description: Once `triggerClaim` runs, the round already knows who actually made it into the winning allocation. Everyone else is refunded and no longer matters to the live board. The problem is that those excluded addresses stay in `roundPartners`, so the contract keeps treating them as part of the round's working set even after the round has moved to `Active`.

```
address[] memory _partners = $.roundPartners[auctionId];
for (uint256 i = 0; i < _partners.length; i++) {
    address p = _partners[i];
    if ($.slots[auctionId][p].allocatedWeight == 0) {
//...
        _creditRefund(p, dep);
    }
}
```

```
address[] memory _partners = $.roundPartners[auctionId];
for (uint256 i = 0; i < n; i++) {
    if ($.slots[auctionId][_partners[i]].allocatedWeight > 0) {
        includedCount++;
    }
}
```

That means the active-round code pays for everybody who ever registered, not just the winners who still shape the board. `submitProposal` has to walk the full historical partner list to rebuild weights, and active `updateSlotVault` checks walk the same list again when a slot holder wants to redirect their vault.

A realistic example is a busy round where a large number of partners register while the auction is still expensive, but only a small set are actually competitive once the claim is triggered. Imagine 150 addresses register during the round, but only 6 end up included in the final allocation. The other 144 are excluded and refunded, yet they remain in `roundPartners`. Later, when one of the winning slot holders needs to update their vault or when the protocol needs to submit the proposal, the contract still iterates over all 150 historical entries to find the 6 that still matter. Nothing is wrong logically, but the cost of active-round operations now scales with stale participation rather than the live allocation.

In a heavily contested round, that overhead can become the difference between a routine active-round update and a transaction that is expensive enough to delay, discourage, or eventually prevent timely proposal handling. The issue is not that excluded partners keep control rights. The issue is that they keep their gas footprint.

Recommendation: When a round becomes `Active`, store a compact winner list and use that list for active-round logic. That keeps proposal assembly and vault-update validation proportional to the actual live board instead of the full registration history. Another valid and smaller mitigation is to cap open-round participant growth more aggressively or add auxiliary indexes so active paths do not need to rescan every excluded partner.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.1.10 Active slot-vault edits can desync proposals

Severity: Medium Risk

Context: CuttingBoardSyndicate.sol#L653-L677, CuttingBoardDutchAuction.sol#L505-L506

Description: In syndicate-owned active rounds, the protocol can keep a manager proposal alive even after the live cutting board has changed. The initial proposal is created at claim time from the then-current weights, but active `SlotNFT` holders are still allowed to mutate the live round by changing their slot vaults afterward. The round's live board is reconstructed from current slot storage, while the already-created manager proposal remains unchanged until it is explicitly cancelled, expires, or gets approved.

```
$.controlManager.proposeCuttingBoard(tokenId, initialWeights);
//...
$.slots[auctionId][originalPartner].vault = newVault;
//...
weights[idx++] = IBeraChef.Weight({
    receiver: slot.vault,
    percentageNumerator: slot.allocatedWeight
});
```

Because `CuttingBoardManager` allows only one active proposal per token, the protocol has no automatic way to refresh that proposal when an active-round vault edit happens. This means `previewWeights()` can describe one board while `getProposal(tokenId)` still exposes an older one. If a keeper approves the stale proposal before anyone cancels or replaces it, the old board is what gets queued upstream, silently discarding the live slot edit. If the stale proposal is not approved, the round still depends on manual cancellation or expiry before a fresh proposal can be submitted.

For example, imagine a syndicate round that claims a validator with two live entries, such as `VaultA = 6000 bps` and `VaultB = 4000 bps`. That claim path immediately creates a manager proposal with those same weights. After the round becomes active, the holder of the winning `SlotNFT` for the `VaultA` share can call the active-round `updateSlotVault(uint256 auctionId, address originalPartner, address newVault)` entrypoint and set `newVault = VaultC`. That call is authorized through current `SlotNFT` ownership, so the live round now wants `VaultC = 6000 bps` and `VaultB = 4000 bps`. The pending manager proposal is still the older `VaultA` / `VaultB` version, though. If a keeper now approves that pending proposal, the protocol queues the stale `VaultA` allocation upstream even though the active round state and `previewWeights()` already reflect `VaultC` instead.

The main impact is an integrity and liveness failure in the active-round update flow. Active slot holders can successfully change the live round state without that change being reflected in the pending approval object that keepers act on.

Recommendation: Keep the pending proposal and the live active-round board coupled to the same source of truth. A direct fix is to make active `updateSlotVault` invalidate or replace any existing proposal for that round's control NFT, so a keeper can never approve stale weights. As defense in depth, `approveCuttingBoard` should also verify that the proposal being approved still matches the current active-round weights for syndicate-owned positions before queueing it upstream. If that coupling is not desired, then active slot edits should be blocked while a proposal is pending.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.1.11 Reauction can overlap a stale active round

Severity: Medium Risk

Context: CuttingBoardDutchAuctionV1_1.sol#L83-L87, CuttingBoardSyndicate.sol#L802-L806, CuttingBoardSyndicate.sol#L861-L864, CuttingBoardDutchAuction.sol#L882-L903

Description: The protocol can start a fresh auction for a validator even while the syndicate still keeps the previous round for that same validator in `Active` state. The gate in `CuttingBoardDutchAuction` only checks whether the validator still has an active auction or a still-valid control NFT. It does not check whether the syndicate still has an active round tied to the older control period.

```
if ($.activeValidatorAuctions[validatorHash] != 0) {
    return false;
}

uint256 tokenId = $.validatorControlTokenId[validatorHash];
if ($.controlNFT.isValid(tokenId)) {
    return false;
}

return true;
```

At the same time, `triggerClaim` permanently advances the round into `Active` and stores the claimed control token id, but there is no matching state transition that retires or invalidates that round once the control NFT naturally expires.

```
$.rounds[auctionId].state = CuttingBoardSyndicateLib.RoundState.Active;
//...
$.rounds[auctionId].tokenId = controlTokenId;
```

That creates a split-brain state around one validator. The auction layer believes the validator is free again as soon as the old control NFT expires, while the syndicate layer still treats the old round as active and still allows active-round semantics to apply to that stale state.

A realistic way to trigger this is straightforward. A keeper starts an auction for validator `V`. The syndicate opens the round, partners register and someone calls `triggerClaim`, which claims the validator and moves the round into `Active`. Time then advances past the old control NFT expiry without anybody explicitly cleaning up the round. At that point `CuttingBoardDutchAuction` sees no live control NFT for `V`, so the keeper can start a brand new auction for the same validator. The protocol now has a new live auction for `V` while the previous syndicate round for `V` is still marked `Active`. Old slot holders can still interact with that stale active round and offchain systems can observe two different lifecycle states for the same validator at once.

Because of this, a single validator can accumulate overlapping lifecycle state across two different auction epochs, which breaks the assumption that one validator maps to at most one live control context at a time. That can mislead keepers, stale proposal handling, active round maintenance, slot-holder rights and any monitoring that treats `Active` round state as proof that the validator is still in the old control period.

Recommendation: Keep validator availability aligned with syndicate round lifecycle, not only with auction and NFT lifecycle. The cleanest fix is to prevent `startCuttingBoardAuction` from reauctioning a validator while any syndicate round for that validator is still `Active`. If reauction after expiry is intended, then the old syndicate round should first be moved into an explicit terminal state so that stale active-round rights and stale `Active` status cannot survive into the next auction epoch. More generally, the protocol should maintain a single authoritative per-validator lifecycle invariant across auction, control NFT and syndicate round state.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.1.12 Inconsistent Buffer Vault Reference Enables Duplicate Vault After Governance Change

Severity: Medium Risk

Context: CuttingBoardSyndicateLib.sol#L354-L359

Description: `CuttingBoardSyndicateLib` maintains two separate paths for assembling the `IBeraChef.Weight[]` array submitted to `CuttingBoardManager`. The first, `weightsFromFill`, is called inside `triggerClaim` and reads the buffer vault receiver directly from the live `$.bufferVault` storage variable. The second, `weightsFromStorage`, is called by `submitProposal` during the Active phase and correctly reads the buffer vault from the per-round snapshot stored in `$.rounds[auctionId].bufferVault` at trigger time. This design inconsistency is the root cause of the vulnerability.

`validateVaultUpdate`, called by the Active-state overload of `updateSlotVault` to prevent duplicate vault entries, also reads the live `$.bufferVault`. These two references diverge the moment governance calls `setBufferVault` while a round is Active. After that call, `validateVaultUpdate` no longer treats the old buffer vault address as forbidden - the live value has already changed. A partner can therefore assign their slot vault to the old buffer vault address, which passes all validation checks. However, when `submitProposal` later calls `weightsFromStorage`, it produces two entries with the same receiver: one from the partner's slot and one from the round snapshot that still records the old address. `CuttingBoardManager._checkForDuplicateReceivers` then reverts every call to `submitProposal` for that round, permanently bricking it. Partners who paid at trigger time are left with an Active round that can never produce a proposal, with their control NFT ticking toward expiry and no refund path available.

Attack Scenario.

1. `triggerClaim` runs with `$.bufferVault = V_old`. Snapshot set to `V_old`.

2. Governance calls `setBufferVault(V_new)`. Now `$.bufferVault = V_new`.
3. A partner calls `updateSlotVault(auctionId, partner, V_old)` (3-arg Active-state variant).
4. `validateVaultUpdate` checks `V_old == $.bufferVault` → `V_old == V_new` → **false**. No other partner has `V_old`. The update succeeds.
5. `submitProposal` calls `weightsFromStorage`, which uses the snapshot `V_old` for the buffer entry **and** finds a partner vault also set to `V_old`. Two entries share the same vault address.
6. `CuttingBoardManager._validateWeights` → `_checkForDuplicateReceivers` catches the duplicate and reverts, permanently blocking proposal submission for this round.

Recommendation: `validateVaultUpdate` should compare the proposed vault against both the live `$.bufferVault` and the round-snapshotted buffer vault when the round is Active, blocking reassignment to either value. To address the underlying design inconsistency, `weightsFromFill` should receive the already-snapshotted buffer vault address as a parameter rather than reading `$.bufferVault` live at call time, ensuring both weight-assembly paths reference the same source of truth for any given round.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.1.13 De-whitelisted Vault Can Permanently Freeze an Active Round

Severity: Medium Risk

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

Description: Once `CuttingBoardSyndicate.triggerClaim` succeeds, partner vaults are fixed in slot storage and the round transitions to Active. From this point, `submitProposal` calls `CuttingBoardSyndicateLib.weightsFromStorage` to assemble the allocation and forwards it to `CuttingBoardManager.proposeCuttingBoard`, which re-validates every receiver address against BeraChef's `isWhitelistedVault` at proposal time rather than at trigger time. If BeraChef governance de-whitelists any vault that was valid at trigger time, `proposeCuttingBoard` reverts on every subsequent call and `submitProposal` becomes permanently bricked for that round.

The only on-chain recovery path is for the holder of the affected partner's SlotNFT to call the three-argument overload of `updateSlotVault` with a currently whitelisted vault. If the SlotNFT has been lost, burned, or is held by an unresponsive address, no rescue is possible: the control NFT ticks toward expiry, no proposal is ever submitted, and partners who paid at trigger time have no refund mechanism - the contract provides no fallback that credits deposits back when an Active round fails to produce any proposal.

Recommendation: Add a governance-gated emergency function that can force-expire an Active round, credit partner refunds proportional to what they paid at trigger time, and transition the round to Expired. Alternatively, cache each vault's `BeraChef` whitelist status at trigger time in the slot storage and use that recorded value during proposal validation, decoupling round liveness from post-trigger changes in `BeraChef` governance.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 `withdrawBuffer` can conflict with open rounds

Severity: Low Risk

Context: CuttingBoardSyndicate.sol#L423-L434

Description: `withdrawBuffer` allows governance to reduce buffer reserves with just an underflow protection. It does not account for already-open rounds that may rely on current buffer levels to remain triggerable.

```
function withdrawBuffer(address to, uint256 amount) external virtual onlyGovernor {
//...
    $.bufferDeposit -= amount;
```

```
    $.paymentToken.safeTransfer(to, amount);
}
```

`triggerClaim` later enforces `bufferDeposit >= bufferRequired` and reverts otherwise.

```
if (fill.bufferRequired > $.bufferDeposit) {
    revert InsufficientBuffer(fill.bufferRequired, $.bufferDeposit);
}
```

For example, assume Alice opens a round and registers a 5000 bps slot at a price where the remaining 5000 bps must be covered by the `bufferVault`. At that point the round is triggerable because `bufferDeposit` holds exactly the amount needed for the buffer side of the fill. If governance then withdraws even 1 wei from the reserve before anyone calls `triggerClaim`, nothing about Alice's slot changes, but the round is no longer executable because the same fill now fails the `InsufficientBuffer` check. A governance withdrawal can therefore convert previously triggerable open rounds into untriggerable rounds without any participant action.

Recommendation: Consider adding a withdrawal safety policy against in-flight open rounds. The strongest option is to reject withdrawals that would break trigger feasibility for any currently open round under present inputs. If full scanning is too expensive, enforce a conservative reserve floor and timelocked withdrawals when open rounds exist.

Infrared Finance: Acknowledged.

Cantina Managed: Acknowledged.

3.2.2 Dead open rounds still accept slot writes

Severity: Low Risk

Context: CuttingBoardSyndicate.sol#L567-L570

Description: Once a round is opened, the main open-state write paths only check whether the round is still marked `Open`. They do not recheck whether the underlying Dutch auction is still live. That means a round can be economically dead while still accepting new slot registrations and deposit increases until someone calls `expireRound`.

```
if (
    $.rounds[auctionId].state
        != CuttingBoardSyndicateLib.RoundState.Open
) revert NotOpen();
```

The only place that synchronizes round state with auction liveness is `expireRound`.

```
if ($.dutchAuction.isAuctionActive(auctionId)) {
    revert AuctionStillLive();
}
```

Consider a simple case where Alice opens a round and waits for more partners. Before anyone triggers, an external bidder claims the auction directly. The syndicate round is now dead in practice, but it still sits in `Open` state until cleanup happens. During that window Bob can still call `registerSlot`, Carol can still call `increaseMaxPrice`, and both transactions succeed even though the round can never win. Their funds are not permanently lost because `exitSlot` is still available and anyone can later call `expireRound`, but the contract still accepts fresh capital and state changes for a round that is already over.

This is mainly a state-consistency problem. It can mislead users and automation into interacting with dead rounds and paying gas for actions that no longer have a path to success.

Recommendation: Recheck auction liveness in the Open-state mutators that are supposed to matter only while the auction can still be won. A direct fix is to require `isAuctionActive(auctionId)` in `registerSlot`, `increaseMaxPrice` and the Open-state `updateSlotVault` path, while still allowing `exitSlot` and `expireRound` after the auction has died so users can unwind cleanly.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.2.3 `setSlotNFT` accepts incompatible contracts

Severity: Low Risk

Context: CuttingBoardSyndicate.sol#L456-L459

Description: `setSlotNFT` accepts and stores any address. The rest of the system then assumes that the configured contract is a `CuttingBoardSlotNFT` instance deployed for this specific syndicate. That assumption is not checked when the configuration is written.

This matters because `CuttingBoardSlotNFT` gates both minting and metadata updates through its immutable `syndicate` address. If governance points the syndicate at a contract that was deployed for a different caller, or even at an arbitrary address with a matching interface shape, the configuration will succeed but the failure will only surface later on the live execution path.

```
function setSlotNFT(address _slotNFT) external virtual onlyGovernor {
    _getStorage().slotNFT = CuttingBoardSlotNFT(_slotNFT);
}
```

```
address public immutable syndicate;
modifier onlySyndicate() {
    if (msg.sender != syndicate) revert NotSyndicate();
    _;
}
```

In practice, that means `triggerClaim` can revert when it tries to mint slot NFTs, and active-round vault updates can fail when they try to synchronize metadata. The round itself may be otherwise valid, but the system becomes unusable because the dependency was accepted without any compatibility check.

Recommendation: Consider validating nonzero candidates inside `setSlotNFT` before saving them. The simplest check is to require `candidate.syndicate() == address(this)`.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.2.4 Trigger preflight can be inaccurate

Severity: Low Risk

Context: CuttingBoardSyndicateLib.sol#L289-L314

Description: `canTrigger` and `effectiveTriggerPrice` are meant to answer a practical question: whether `triggerClaim` would succeed for the round as it exists right now. Today they do not fully answer that question. They model the fill, buffer and entry-count constraints, but they do not cover every hard precondition enforced on the execution path.

One concrete gap is `slotNFT` configuration. The helper checks can report that a round is ready, while `triggerClaim` still reverts because `slotNFT` has not been set. That creates a false-positive readiness signal for keepers, bots and dashboards that rely on the view layer to decide when to submit the transaction.

```
function canTrigger(...) external view returns (bool) {
//...
    if (fill.bfWeight > 0 && $.bufferVault == address(0)) return false;
    return $.bufferDeposit >= fill.bufferRequired;
}
```

```
CuttingBoardSlotNFT _slotNFT = $.slotNFT;
if (address(_slotNFT) == address(0)) revert SlotNFTNotSet();
```

Recommendation: Consider moving the trigger checks into a shared validation function and reuse it from both the helper views and `triggerClaim`. At minimum, the helpers should include `slotNFT` readiness alongside the existing fill, buffer and entry-limit checks so the view path stays aligned with the execution path.

Infrared Finance: Acknowledged. SlotNFT is assumed to be set at deployment (included in deploy script: DeployCuttingBoardSyndicate.s.sol#L123-L126)

Cantina Managed: Acknowledged.

3.2.5 `setMinSlotWeight` can exceed live `maxWeightPerVault` and block registrations

Severity: Low Risk

Context: CuttingBoardSyndicate.sol#L441-L447

Description: `setMinSlotWeight` checks that the new value is a nonzero divisor of 10000, but it does not check whether that value is still usable under the current `BeraChef` limits.

That leaves a simple configuration failure mode. Governance can set `minSlotWeight` to a value that is formally valid for the syndicate, but larger than the live `maxWeightPerVault` enforced by `chef`. Once that happens, the contract still accepts the configuration, but no future registration can satisfy both sets of rules at the same time.

```
function setMinSlotWeight(uint96 minWeight) external virtual onlyGovernor {
    if (minWeight == 0 || 10000 % minWeight != 0) {
        revert InvalidMinSlotWeight();
    }
    $.minSlotWeight = minWeight;
}
```

The incompatibility only becomes visible later inside `registerSlot`, which requires the weight to be both a multiple of `minSlotWeight` and no greater than `chef.maxWeightPerVault()`. If `minSlotWeight` is above the cap, there is no positive weight that passes both checks.

```
if (weight == 0 || weight % $.minSlotWeight != 0) revert InvalidWeight();
if (weight > $.chef.maxWeightPerVault()) revert InvalidWeight();
```

For example, suppose the live `BeraChef` cap is 5000 bps (`chef.maxWeightPerVault()` returns 5000) and governance updates `minSlotWeight` to 10000 because 10000 is still a valid divisor of the full board. Alice then tries to register a 5000 bps slot for her vault. That fails because 5000 is not a multiple of the new minimum. If she instead tries 10000 bps to satisfy the divisibility rule, that fails because it exceeds the live BeraChef cap. At that point Alice, and every other participant, is locked out of opening new slots until governance changes the setting again.

Recommendation: Validate the value against the live BeraChef cap when `setMinSlotWeight` is called, for example by requiring `minWeight <= chef.maxWeightPerVault()`. If the external cap can move independently over time, it is also worth treating that relationship as an operational invariant and monitoring for drift so the system does not end up in an impossible configuration later.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.2.6 `effectiveTriggerPrice` can underestimate max price

Severity: Low Risk

Context: CuttingBoardSyndicateLib.sol#L336-L350

Description: `effectiveTriggerPrice` is presented as the highest price where `triggerClaim` would succeed, but the implementation evaluates only discrete candidate prices built from partner breakpoints (`maxPricePerBps * 10000`). It does not search prices between those candidates.

```
for (uint256 i = 0; i < n; i++) {
    uint256 p = breakpoints[i] * 10000;
    FillResult memory fill = _computeFillInternal($, auctionId, p);
//...
    if ($.bufferDeposit >= fill.bufferRequired) return p;
}
```

Feasibility can still change between sampled points because `bufferRequired` is affected by integer division and residual accounting. A sampled breakpoint can fail while a slightly lower unsampled price inside the same interval would pass. In that case, the helper falls through to a lower breakpoint and returns

a value below the true maximum feasible price. The result is conservative but can mislead operators who treat the output as exact and can delay keeper execution.

For example, assume the sampled breakpoints are 1200 and 1000 and the current `bufferDeposit` is 100 tokens. At sampled price 1200, the computed `bufferRequired` is 101 so the check fails, but at an unsampled intermediate price like 1150 (same eligibility interval) `bufferRequired` drops to 99 and `triggerClaim` would actually succeed; since the helper never tests 1150, it skips directly to 1000 and returns that lower value, making keepers believe they must wait for a larger price drop than truly required.

Recommendation: If this helper is intended to return the exact maximum, scan each breakpoint interval and use a bounded binary search with the same feasibility checks used by `triggerClaim`. If exactness is not required, rename and document the function as a conservative lower-bound estimator so integrators do not rely on it as an exact maximum trigger price.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.2.7 Invalidated NFTs remain transferable

Severity: Low Risk

Context: CuttingBoardNFT.sol#L217-L223

Description: `CuttingBoardNFT` separates ERC-721 ownership from the underlying control right. When a token is invalidated, the contract marks the right as inactive, but it does not burn the NFT or stop it from being transferred. The token can still move like an ordinary ERC-721 even though it no longer gives the holder any usable control over the validator.

```
function invalidate(uint256 tokenId) external virtual onlyManager {
    if (tokenId == 0 || tokenId >= _nextTokenId) revert InvalidTokenId();
    _tokenRights[tokenId].active = false;
    emit ControlNFTInvalidated(tokenId);
}
```

The problem is not just that the token still exists. The metadata path does not make the invalid state obvious either. `tokenURI` returns the same base URI regardless of whether the token is still valid, so generic NFT interfaces can present a revoked token exactly like a live one.

```
function tokenURI(uint256 tokenId) public view virtual override returns (string memory) {
    if (_ownerOf[tokenId] == address(0)) revert InvalidTokenId();
    return _baseTokenURI;
}
```

A realistic example is that a validator control NFT is invalidated after expiry or through an administrative action, but the current holder later transfers or sells it through a normal NFT venue. The buyer sees a transferable ERC-721 and receives the token successfully. Only later, when they try to use it in a protocol flow that checks `isValid`, they discover that the NFT no longer carries any live control right. At that point the sale may already be settled even though the thing that mattered economically was gone before the transfer.

This is mainly a market-integrity and user-protection issue. The harm comes from tradability and apparent ownership continuing to signal value after the control right itself has already been revoked.

Recommendation: Make the NFT behavior match the right it represents. The cleanest fix is to stop revoked or expired control NFTs from being transferred at all, so a token with no usable control right cannot keep circulating as if it were still live.

If the team wants these tokens to remain transferable for historical or collectible reasons, then the invalid state needs to be made obvious everywhere a buyer would look. In that version, metadata should expose whether the token is still valid and every official UI or integration should treat `isValid` as the source of truth for control rights rather than token ownership alone.

Infrared Finance: Acknowledged. `CuttingBoardNFT` is already deployed and not upgradeable, without a more serious migration plan with other contracts. Will make notes for user guide that NFT UX is sub-optimal by means of no dynamic tokenURI and no burn / transfer halt on expiry.

Cantina Managed: Acknowledged.

3.2.8 Buffer vault set can conflict with open rounds

Severity: Low Risk

Context: CuttingBoardSyndicate.sol#L409-L417

Description: `setBufferVault` only checks whether the new vault is whitelisted. It does not check whether that vault is already being used by a partner in one of the rounds that is still open.

That means governance can accidentally create a conflict after users have already registered. A round may start in a healthy state, with each partner using a unique receiver. Later, governance can point the global `bufferVault` at one of those same receivers. Nothing fails at the time of the config change, but the round is now carrying a hidden collision into its later claim path.

```
function setBufferVault(address vault) external virtual onlyGovernor {
    if (vault != address(0) && !$.chef.isWhitelistedVault(vault)) {
        revert VaultNotWhitelisted();
    }
    $.bufferVault = vault;
}
```

```
if (bfWeight > 0) {
    weights[count] = IBeraChef.Weight({
        receiver: $.bufferVault,
        percentageNumerator: bfWeight
    });
}
```

For example, imagine that Alice registers in an open round using `VaultA`. At that point the round is fine, because no other partner is using `VaultA` and the current buffer vault is different. Later, governance updates the global buffer vault and also chooses `VaultA`. If the round eventually needs any non-zero buffer allocation, the claim logic will try to build one weight entry for Alice's slot and another weight entry for the buffer, both pointing to `VaultA`. Downstream validation expects receivers to be unique, so the claim can now fail even though Alice's registration was perfectly valid when she made it.

Recommendation: Do not let `setBufferVault` accept a receiver that is already reserved by a partner in any still-open round. The simplest fix is to reject the update when a conflict already exists. If scanning open rounds directly is too expensive, maintain a lightweight index of vaults reserved by open rounds and use that index to block unsafe buffer-vault rotations before they land.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.2.9 `minimumPricePerBps()` Integer Truncation Allows Zero-Deposit Registration

Severity: Low Risk

Context: CuttingBoardSyndicate.sol#L322-L324

Description: `CuttingBoardSyndicate.registerSlot` enforces a minimum price per basis point by comparing the partner's `maxPricePerBps` argument against `dutchAuction.minimumPrice() / 10000`. When the auction's minimum price is smaller than 10,000 token units, Solidity integer division truncates the result to zero. The guard condition then becomes equivalent to requiring `maxPricePerBps` to be greater than zero, which is trivially bypassed by submitting `maxPricePerBps = 0`. A zero-price registration computes a deposit of `0 × weight = 0`, so `safeTransferFrom` collects nothing and the registration is entirely free.

A partner registered at zero price is permanently ineligible: Dutch auctions do not reach a clearing price of zero, so `computeFill` will never include them. They nonetheless occupy a permanent slot in `roundPartners` and are iterated by every O(n) operation in `triggerClaim`, `expireRound`, the duplicate-vault scan in `registerSlot`, and `_removePartner`. The only removal path is for the partner to call `exitSlot`, which refunds zero tokens at the cost of gas alone. Combined with the unbounded registration

cap described in M-4, zero-price registrations make a gas DoS against round operations completely free to execute.

Recommendation: Guard against a zero result from the integer division before applying the comparison in `registerSlot`. The minimum acceptable value for `maxPricePerBps` should be at least 1 regardless of what the division yields, ensuring that a deposit of zero is never possible. The same correction should be applied to the `minimumPricePerBps()` view function so it returns an accurate floor to off-chain callers.

Infrared Finance: Acknowledged.

Cantina Managed: Acknowledged.

3.2.10 Unbounded `roundPartners` Array Enables Per-Round Gas DoS

Severity: Low Risk

Context: CuttingBoardSyndicateLib.sol#L99

Description: `CuttingBoardSyndicate.registerSlot` applies no limit on the number of partners that can register for a given auction. The `roundPartners` array for any given round can therefore grow without bound, and every major round operation iterates it in full. `triggerClaim` performs an O(n) exclusion pass over all registered partners and an O(n log n) sort via `computeFill` in `CuttingBoardSyndicateLib`; `expireRound` issues O(n) refund credits followed by O(n) slot deletions in `_clearSlots`; `registerSlot` itself executes an O(n) duplicate-vault scan on each new registration; and `_removePartner`, called by `exitSlot`, performs an O(n) linear search. As the partner count grows, all of these operations approach the block gas limit.

The existing `TooManyPartners` guard fires at trigger time and counts only partners that are eligible to be included in the fill against BeraChef's `maxNumWeightsPerRewardAllocation`. A round populated with many registered-but-ineligible partners passes this check while still making `triggerClaim` and `expireRound` prohibitively expensive. When combined with M-2, which allows zero-cost registrations when the auction's minimum price truncates to zero, mounting this attack requires no capital beyond the gas cost of the registration calls themselves.

Recommendation: Enforce the registration cap eagerly in `registerSlot` by rejecting any new registration that would push the `roundPartners` count beyond BeraChef's `maxNumWeightsPerRewardAllocation`. The existing `TooManyPartners` error is already defined and appropriate for this guard, making the limit self-documenting and consistent with the check that already exists at trigger time.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.3 Informational

3.3.1 `roundTotalWeight` is unused for protocol logic

Severity: Informational

Context: CuttingBoardSyndicateLib.sol#L101-L102

Description: The syndicate keeps a per-round `roundTotalWeight` mapping up to date as partners register, exit and as rounds are cleared. That value is also exposed through `getRoundTotalWeight()`. Within the scoped production implementation, however, this state does not appear to drive any protocol decision. The claim path does not use it, slot validation does not use it and proposal assembly does not use it. It is maintained entirely in parallel with the real source of truth, which is the per-partner slot set itself.

```
$.roundPartners[auctionId].push(msg.sender);
unchecked {
    $.roundTotalWeight[auctionId] += weight;
}
```

```
unchecked {
    $.roundTotalWeight[auctionId] -= slot.weight;
}
```

```
delete $.roundPartners[auctionId];
$.roundTotalWeight[auctionId] = 0;
```

That makes `roundTotalWeight` look like telemetry rather than load-bearing state. If that is the intent, keeping it on-chain still has a cost. Every registration, exit and round cleanup pays additional write overhead to maintain a value that is not consumed by the live protocol. It also creates another piece of state that future refactors must remember to keep in sync, even though correctness does not depend on it.

The impact is limited to maintainability and gas efficiency. The main downside is carrying extra state on hot paths without a corresponding behavioral benefit.

Recommendation: If the protocol only needs this value for visibility, consider removing `roundTotalWeight` from storage and deriving it off-chain from `SlotRegistered` and `SlotExited` events, or from the slot set itself in view-only tooling. If the team wants to keep the getter for operator ergonomics, document clearly that `roundTotalWeight` is intentional telemetry-only state so its ongoing storage cost is a conscious tradeoff rather than an accidental leftover.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.3.2 Unused Strings import

Severity: Informational

Context: CuttingBoardNFT.sol#L6

Description: `CuttingBoardNFT` imports `Strings` from OpenZeppelin and declares `using Strings for uint256`, but the contract does not call any `Strings` helpers. That leaves a dead import and an unused `using` directive in the file.

```
import {Strings} from "@openzeppelin/contracts/utils/Strings.sol";

contract CuttingBoardNFT is ERC721, Owned {
    using Strings for uint256;
```

Recommendation: Remove the `Strings` import and the `using Strings for uint256` directive.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.3.3 Global refund sweep can exceed gas limits

Severity: Informational

Context: CuttingBoardSyndicate.sol#L986-L996

Description: `claimAllRefunds` iterates over the full historical `refundees` array and cost scales with total historical participants, not with currently nonzero refund balances. The refundee registry only grows and is never compacted, so sweep complexity is monotonically increasing over protocol lifetime.

```
address[] storage all = $.refundees;
uint256 n = all.length;
for (uint256 i = 0; i < n; i++) {
    address user = all[i];
    uint256 amount = $.pendingRefunds[user];
    if (amount > 0) { ... }
}
```

```
if (!$.isRefundee[user]) {
    $.isRefundee[user] = true;
    $.refundees.push(user);
}
```

Per-user claims remain possible, but the global sweep path can become non-callable under practical gas limits as participation grows. The impact is operational liveness degradation for keeper-style bulk refund workflows.

Recommendation: Consider introducing pagination or bounded batching for bulk refunds so each call has predictable upper gas cost. As an alternative, maintain an active-debt index keyed to users with nonzero pending refunds, and iterate that bounded set instead of the historical participant set.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.3.4 Berachain update changes syndicate semantics

Severity: Informational

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

Description: The upcoming Berachain `BeraChef` update referenced in commit `136ab44a2de08683fdf12016e9c0a658ddd8a13d` changes the meaning of an active custom cutting board. The new implementation adds `rewardAllocationInactivityBlockSpan`, introduces a new `RewardAllocatorFactory` and changes `getActiveRewardAllocation` so that a validator-specific allocation is no longer treated as effective indefinitely until it is explicitly replaced. Instead, a validator-specific allocation can become inactive purely because it has not been refreshed within a configured block span. Once that happens, `BeraChef` stops using the validator's custom allocation and falls back first to a factory-managed baseline allocation and only falls all the way to the default allocation if that baseline is missing or invalid. The commit therefore introduces a three-layer resolution model of `custom` → `baseline` → `default`, not a simple `custom` → `default` switch. It also does not hardcode seven days as the inactivity window is configurable and should not be treated as a literal contract constant.

This matters to the current syndicate design because the syndicate state machine was built around a simpler invariant. The syndicate opens a round, computes a winning composition at `triggerClaim`, stores each winner's `allocatedWeight`, receives the `CuttingBoardNFT` and later reconstructs the same board through `weightsFromStorage` when `submitProposal` is called. Partners can change destination vaults during the active period, but the stored slot weights still represent a single intended composition for that round. In that model, the round being `Active`, the slot rights still being valid, and the validator still running the syndicate-composed board all feel like the same fact viewed through different interfaces.

After the Berachain update, those facts can drift apart. A round can remain `Active`, the control NFT and per-slot NFTs can still be valid, the stored slot allocation can still describe the intended winning composition and yet Berachain may already be routing rewards according to the factory baseline or the default allocation because the last approved syndicate board crossed the inactivity boundary without a refresh. The syndicate can still reconstruct the intended board, but it can no longer assume that this board is what Berachain is currently honoring. In other words, the current syndicate design stops being only a round-allocation state machine and becomes a liveness-sensitive control loop. The protocol may need to refresh an unchanged composition during a single active round just to preserve the status quo.

The syndicate is also affected differently from a direct auction winner. A direct winner currently lacks a clean keeper-only path to recreate an unchanged board, but the syndicate is structurally better positioned because it owns the `CuttingBoardNFT` for active rounds and already has a path to call `CuttingBoardManager.proposeCuttingBoard` through `submitProposal`. That means the syndicate can probably refresh its own board without requiring each partner to sign again. At the same time, this path is currently implicit rather than intentional. `submitProposal` is permissionless, `CuttingBoardManager` only allows one live proposal at a time and the system does not model periodic refresh as a first-class responsibility. As a result, the upcoming Berachain change turns proposal cadence, spam tolerance, cancellation behavior and keeper approval timing into design questions for the syndicate rather than mere operational details.

The fallback model itself also matters for user expectations and accounting semantics. If the next version of the syndicate assumes that inactivity means ”the board reverted to default”, that model can be wrong whenever Berachain configures a non-default baseline allocation in the new factory. An active round may therefore have at least four meaningful states: the desired syndicate-composed board, the last approved validator-specific board, the effective Berachain allocation and the fallback layer currently in force. If those states are not modeled separately, the local round can appear healthy while the external reward-routing behavior has already changed. The issue is primarily informational because it describes an upcoming

upstream semantic change rather than a present exploit in the current deployment, but it has direct consequences for how the next syndicate version should define ”active”, ”up to date”, and ”control”.

Recommendation: The next syndicate version should be designed around an explicit refresh model rather than assuming that the last approved composition stays effective until NFT expiry. The main design choices that should be settled up front are:

1. Decide what guarantee the syndicate is supposed to provide. If `Active` is meant to guarantee that the syndicate-composed board is still the effective Berachain board, then the design needs a refresh mechanism and corresponding observability. If `Active` only means that the syndicate still has the right to restore or change the board, that weaker guarantee should be made explicit in docs, views and keeper runbooks.
2. Decide who is allowed to refresh an unchanged board. The clean options are a permissionless refresh path through the syndicate, a keeper-only refresh path, a governance-controlled refresh path, or a more restrictive partner-authorized path. This decision should be made together with the existing `submitProposal` griefing surface, because a periodic refresh requirement makes proposal spam materially more important than it is today.
3. Decide what state should be treated as the source of truth for a refresh. The candidates are the round's current `weightsFromStorage` reconstruction, the most recently approved board, or Berachain's own last validator-set board. The choice should be explicit, because ”refresh the current desired board” and ”refresh the last board Berachain saw from this validator” are not always the same once inactivity fallback exists.
4. Add explicit state or views for desired-versus-effective allocation. The next version should be able to distinguish between the syndicate's intended composition and the board Berachain is actually using. In practice this likely means extending the Berachain-facing read surface so the protocol can reason about inactivity span, last validator-set allocation and whether fallback is currently hitting baseline or default.
5. Decide how `allocationDuration`, proposal validity, queue delay, and inactivity span should interact. A robust design should not rely on a nominal block time alone. It should either enforce safe margins at configuration time or build automation around a conservative refresh threshold that leaves room for queueing, keeper approval and activation before the inactivity boundary is crossed.
6. Decide whether refresh should be first-class in the syndicate state machine. A clean redesign may warrant an explicit refresh operation, an explicit ”stale but recoverable” status, or an epoch-like heartbeat model during `Active` rounds. Relying on the existing one-off `submitProposal` path may work operationally, but it leaves the most important new responsibility as an implicit side effect instead of a modeled protocol behavior.
7. Add Berachain-specific integration tests before locking the new design. At minimum the test matrix should cover `baseline == default`, `baseline != default`, inactivity shorter than control duration, inactivity longer than control duration, block-time skew, repeated same-weight refreshes, and the case where the round remains active while Berachain has already fallen back away from the syndicate-composed board.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.3.5 `triggerClaim` Hard-Reverts When `slotNFT` Is Not Set, Contradicting Documentation

Severity: Informational

Context: CuttingBoardSyndicate.sol#L451

Description: In `CuttingBoardSyndicate`, the `triggerClaim` function unconditionally reverts with `SlotNFTNotSet` if `$.slotNFT` is `address(0)`. This directly contradicts the NatSpec on `setSlotNFT`, which explicitly states that setting the address to zero disables SlotNFT minting, implying that operating without a configured SlotNFT is a supported and intentional mode.

Recommendation: Document the expected behavior when the `slotNFT` is set to `address(0)`.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.

3.3.6 `tokenURI` Returns Identical URI for All Tokens in Both NFT Contracts

Severity: Informational

Context: CuttingBoardNFT.sol#L237-L238

Description: Both `CuttingBoardSlotNFT` and `CuttingBoardNFT` override `tokenURI` by returning the bare `_baseTokenURI` string without appending the token ID. Every token in each collection resolves to exactly the same metadata URI. Wallets, marketplaces, and off-chain tooling relying on the standard ERC-721 metadata interface cannot distinguish individual tokens from one another, making it impossible to determine which slot rights or control rights a given NFT represents without querying `getSlotRights` or `getControlRights` directly on the contract.

Recommendation: Append the token ID to the base URI inside `tokenURI` using `Strings.toString(tokenId)`, following standard ERC-721 metadata practice so each token resolves to its own endpoint.

Infrared Finance: Acknowledged.

Cantina Managed: Acknowledged.

3.3.7 `safeApprove` Incompatible With Tokens That Prohibit Non-Zero→Non-Zero Allowance

Severity: Informational

Context: CuttingBoardSyndicate.sol#L855-L857

Description: In `CuttingBoardSyndicate.triggerClaim`, the contract grants the Dutch auction contract an allowance of `currentPrice` tokens by calling `$.paymentToken.safeApprove(address($.dutchAuction), currentPrice)`, then resets the allowance to zero after `claimCuttingBoardControl` returns. Solmate's `safeApprove` is a thin wrapper over the raw ERC-20 `approve` call and does not pre-clear the existing allowance before setting a new one. Tokens such as USDT revert when `approve` is called with a non-zero value if the spender already has a non-zero allowance outstanding.

Under the current implementation this is not directly exploitable: a failed `triggerClaim` reverts all state including the approval, so no residual allowance from a prior failed call can persist to the next invocation. However, the pattern creates a latent fragility. Any future code path that leaves a partial allowance in place - such as a mid-execution revert in an upgrade that does not fully unwind the approval - would cause every subsequent `triggerClaim` call to revert unconditionally for USDT-like payment tokens, with no recovery mechanism short of a governance action to manually clear the spender's allowance.

Recommendation: Replace the `safeApprove(nonzero)` call with an explicit reset to zero immediately beforehand, or migrate to OpenZeppelin's `forceApprove`, which performs the reset atomically and is designed for compatibility with non-standard ERC-20 approval semantics.

Infrared Finance: Acknowledged.

Cantina Managed: Acknowledged.

3.3.8 Duplicated `computeFill` Logic

Severity: Informational

Context: CuttingBoardSyndicateLib.sol#L130

Description: `CuttingBoardSyndicateLib` contains two independent implementations of the same fill algorithm: `computeFill`, an external function called via DELEGATECALL from `CuttingBoardSyndicate.triggerClaim` and `previewFillAt`, and `_computeFillInternal`, a private function consumed by `canTrigger` and `effectiveTriggerPrice`. Both implement the identical partner-selection, weight-allocation, and buffer-weight logic. Any change to the fill algorithm - whether a bug fix, rounding rule, or eligibility condition - must be applied to both functions identically. A divergence between them would cause `canTrigger` and `effectiveTriggerPrice` to silently return results inconsistent with what `triggerClaim` would actually execute, breaking keeper automation and off-chain readiness checks without any on-chain error signal. The duplication was introduced to avoid the DELEGATECALL overhead in view-only helpers, but it places the most security-critical logic in the system under a permanent maintenance burden.

Recommendation: Evaluate whether the DELEGATECALL overhead in the view helpers justifies maintaining two copies of the fill algorithm indefinitely. If the performance trade-off is acceptable, consolidating on a single shared implementation eliminates the divergence risk entirely. If the duplication is kept, add explicit cross-reference comments in the NatSpec of both functions so that any future author modifying one is reminded to apply the identical change to the other.

Infrared Finance: Fixed in PR 657.

Cantina Managed: Fix verified.