CANTINA

Kintsu Monad Contracts Core

Security Review

Cantina Managed review by:

Kaden, Lead Security Researcher
Rvierdiiev, Security Researcher

October 15, 2025

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 Critical Risk . . . 4
  - 3.1.1 Accounting error allows for arbitrary share price increase . . . 4
- 3.2 High Risk . . . 4
  - 3.2.1 Accounting error in `unbondDisableNode` results in locked asset tokens . . . 4
- 3.3 Medium Risk . . . 5
  - 3.3.1 `instantUnlock()` is not protected with `whenNotPaused` modifier . . . 5
  - 3.3.2 Sandwich `compound()` to make instant profit . . . 5
- 3.4 Low Risk . . . 6
  - 3.4.1 `syncStaking` should be called before relevant functions . . . 6
  - 3.4.2 Rounding down unbonding amount may cause temporary DoS . . . 6
  - 3.4.3 Incorrect withdrawal delay . . . 8

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

Kintsu is a Composable Liquid Staking Protocol aiming to boost the GDP of DeFi by allowing users to participate in on-chain activities while also benefitting from the yield-bearing staking that secures the blockchain.

From Sep 29th to Oct 2nd the Cantina team conducted a review of monad-contracts-core on commit hash f379f21e. The team identified a total of **7** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	1	1	0
High Risk	1	1	0
Medium Risk	2	2	0
Low Risk	3	3	0
Gas Optimizations	0	0	0
Informational	0	0	0
Total	**7**	**7**	**0**

3 Findings

3.1 Critical Risk

3.1.1 Accounting error allows for arbitrary share price increase

Severity: Critical Risk

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

Description: When we deposit to the protocol via `deposit`, we increment both `totalPooled` and `batchDepositRequests[currentBatchId].assets` by the amount of asset tokens deposited:

```
totalPooled += uint96(msg.value);

batchDepositRequests[currentBatchId].assets += uint96(msg.value);
```

If we then `instantUnlock` to withdraw this deposit before `submitBatch` is called, and thus before the asset tokens are staked, we will decrement `batchDepositRequests[currentBatchId].assets` accordingly, but we do not decrement `totalPooled`:

```
// Remove instantly redeemed assets from current batch
batchDepositRequest.assets = _batchAssets - spotValue;
```

As a result, any deposits that are instantly unlocked will break the accounting of `totalPooled`. This has a significant impact due to the fact that this variable is used for computing the share price:

```
function convertToAssets(uint96 shares) public view returns (uint96 assets) {
    uint256 _totalShares = totalShares();
    if (_totalShares == 0) {
        // This happens upon initial stake
        // Also known as 1:1 redemption ratio
        assets = shares;
    } else {
        assets = uint96(uint256(shares) * uint256(totalPooled) / _totalShares);
    }
}
```

Since `totalPooled` is the numerator for the share price calculation, an attacker can arbitrarily increase the share price by repeatedly depositing and instant unlocking asset tokens. This attack can be leveraged to increase the share price such that their shares can be exchanged for the full amount of asset tokens in the system, draining the system entirely.

Recommendation: Decrement `totalPooled` by the `spotValue` in `instantUnlock`:

```
  // Remove instantly redeemed assets from current batch
  batchDepositRequest.assets = _batchAssets - spotValue;

+  totalPooled -= spotValue;
```

Kintsu: Fixed in commit c7abc87.

Cantina Managed: Fixed as recommended.

3.2 High Risk

3.2.1 Accounting error in `unbondDisableNode` results in locked asset tokens

Severity: High Risk

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

Description: In `unbondDisableNode`, we decrement `totalPooled` by the stake removed from the node being unbonded:

```
// We also need to remove the unbonded amount from totalPooled
// as it's no longer "pooled" for staking.
// The funds are now in a "pending unbond" state.
totalPooled -= _staked;
```

In `sweepForced`, which is called afterwards to withdraw the unbonded amount, we increment `batchDepositRequests[currentBatchId].assets` accordingly, but we don't increment `totalPooled`:

```
uint256 amountWithdrawn = address(this).balance - balanceSnapshot;
if (amountWithdrawn > 0) {
    batchDepositRequests[currentBatchId].assets += uint96(amountWithdrawn);
}
```

As a result, any time we use this pattern to unbond disabled nodes, we will unintentionally decrement `totalPooled`, reducing the share price, thereby reducing the amount of asset tokens which can be withdrawn, ultimately causing asset tokens to be locked in the protocol.

Recommendation: Remove the `totalPooled` decrementation from `unbondDisableNode` as these tokens never leave the system:

```
- // We also need to remove the unbonded amount from totalPooled
- // as it's no longer "pooled" for staking.
- // The funds are now in a "pending unbond" state.
- totalPooled -= _staked;
```

Kintsu: Fixed in 2ad4477.

Cantina Managed: Fixed as recommended.

3.3 Medium Risk

3.3.1 `instantUnlock()` is not protected with `whenNotPaused` modifier

Severity: Medium Risk

Context: StakedMonad.sol#L284

Description: When the protocol is in a paused state, almost all functions are intentionally disabled to preserve system integrity. However, the `instantUnlock()` function is not protected with the `whenNotPaused` modifier. This allows withdrawals to be executed even while the protocol is paused, which breaks the pause invariant and could lead to inconsistent system behavior during emergencies.

Recommendation: Protect the `instantUnlock()` function with the `whenNotPaused` modifier to ensure withdrawals cannot bypass the global pause mechanism.

Kintsu: Fixed in commit 1110bdc.

Cantina Managed: Fixed as recommended.

3.3.2 Sandwich `compound()` to make instant profit

Severity: Medium Risk

Context: StakedMonad.sol#L285

Description: If the `isInstantUnlockEnabled` variable is set to `true`, users can redeem via the `instantUnlock()` function. This function normally charges a fee, but certain users are exempt from fee payment. Such a user can deposit and withdraw in the same block without paying any fees.

This enables a flashloan-based attack:

- Borrow funds with a flashloan.
- Sandwich the `compound()` function with a `deposit()` and an `instantUnlock()`.
- Instantly withdraw without fees, earning a disproportionate share of rewards.
- Repay the flashloan while keeping the profit.

As a result, the attacker could capture a significant portion of rewards with no real risk.

Recommendation: Store the exchange rate of the deposit and only allow instant unlocking at the same exchange rate. This means that old deposits won't get any compounded rewards, even if they deserve them and are not executing this exploit. Effectively this becomes a cancel deposit mechanism and this should be clarified with the function name/docs if implemented. This also would require further consideration if slashing becomes relevant as it will allow users to avoid slashing.

Kintsu: Fixed in commit d3b701a by removing `instantUnlock` entirely.

Cantina Managed: Fix confirmed. Removing `instantUnlock` prevents this attack vector.

3.4 Low Risk

3.4.1 `syncStaking` should be called before relevant functions

Severity: Low Risk

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

Description: The `syncStaking` function is used to socialize slashing across all shareholders, updating slashed nodes' `node.staked` values as well as `totalPooled`:

```
for (uint256 i; i < n; ++i) {

    // ...

    if (newStake > oldStake) {
        // Compound
        rewards += newStake - oldStake;
        node.staked = newStake;
    } else if (newStake < oldStake) {
        // Slash
        slashed += oldStake - newStake;
        node.staked = newStake;
    }
}

if (rewards > 0 || slashed > 0) {
    totalPooled = totalPooled + rewards - slashed;
}
```

As such, the timing of this function being called is relevant to any logic which depends on accurate values for either `node.staked` or `totalPooled`.

For example, if we call `requestUnlock` before a slash is socialized via `syncStaking`, the `spotValue` which we will be able to withdraw is greater than it would be had we done so afterwards, even if the slash had already occurred and just not yet been synced.

Similarly, when `submitBatch` is called, we either bond or unbond amounts for nodes according to their currently staked amounts. If these staked amounts are out of sync, we will bond or unbond incorrect amounts.

Recommendation: Call `syncStaking` at the start of any function which depends on accurate values for either `node.staked` or `totalPooled`, including:

- `deposit`.
- `instantUnlock`.
- `requestUnlock`.
- `submitBatch`.
- `unbondDisableNode`.
- `getInstantUnlockableShares`.

Kintsu: With additional communications from Monad that slashing is not implemented as traditional "remove your balance" slashing, and instead validators are kicked out on the leader schedule if something happens. The `syncStaking()` function has been removed in commit 9d0ce01 and we can revisit socialization of slashing when/if those details are added.

Cantina Managed: Slashing socialization will no longer be needed and the `syncStaking` function has been removed accordingly.

3.4.2 Rounding down unbonding amount may cause temporary DoS

Severity: Low Risk

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

Description: In `submitBatch`, if we are withdrawing more than we are depositing, we will unbond the difference from nodes and pass the dust which is not unbonded to the next batch:

```
} else if (batchWithdrawRequest.assets > batchDepositRequest.assets) {
    // Net egress
    uint96 unbondedAmountEvm = _doUnbonding(batchWithdrawRequest.assets, batchDepositRequest.assets,
    ↪  _totalPooled);
    uint96 dust = batchWithdrawRequest.assets - batchDepositRequest.assets - unbondedAmountEvm;
    if (dust > 0) {
        batchWithdrawRequests[_currentBatchId + 1].assets = dust;
        _totalPooled += dust;
    }
```

In case some dust amount is not unbonded, we may encounter a temporary DoS. Generally this will prevent the last withdrawer from redeeming their assets since there likely will not be enough asset tokens available in the contract to complete their transfer. This may also impact other functions, e.g. if a deposit takes place then the final withdrawer redeems, we may not be able to stake all the deposited funds in the next `submitBatch` call.

This dust amount should ultimately be withdrawn in the next batch, however, it's worth noting that there may still be additional dust not withdrawn from the next batch, repeating the issue.

Recommendation: We can approach this in a couple different ways.

One option is to simply include documentation that this is a possible issue and manually transfer asset tokens into the contract as necessary, perhaps even initially transferring in enough tokens that it should never be an issue.

Another option is to round up the amount being unbonded. This way any dust will be a surplus of assets available to redeem. Although it's important to consider some necessary changes that would need to be made to safely support this change.

Firstly, the `dust` computation would need to be modified as this change would cause it to underflow:

```
- uint96 dust = batchWithdrawRequest.assets - batchDepositRequest.assets - unbondedAmountEvm;
+ int96 dust = int256(batchWithdrawRequest.assets) - int256(batchDepositRequest.assets) -
↪  int256(unbondedAmountEvm);
```

Secondly, the `unbondAmount` in `_doUnbonding` would have to be limited to not exceed `_nodes[i].staked` to avoid underflow:

```
- uint96 unbondAmount = uint96(phase1Amount + phase2Amount);
+ uint96 unbondAmount = uint96(phase1Amount + phase2Amount) > _nodes[i].staked ? _nodes[i].staked :
↪  uint96(phase1Amount + phase2Amount);
```

Note: The above change may be a valuable safety feature to add regardless.

Additionally, it may be wise to include a function to manually withdraw or add dust to the next batch deposit.

Kintsu: I've added comments providing more insight into how dust works and the temporary scenario where redemptions could exceed available funds.

To prevent this issue, the deployment script transfers 0.01 MON (1e16 wei) on deployment which will cover any dust for the lifespan of the protocol. If this is ever exhausted, adding more can be done via transfer to efficiently mitigate the scenario by any address.

See commit c1e95d0.

As for ensuring that MON is never overwithdrawn from a node, the two phase withdraw algorithm in _doUnbonding() accomplishes this by first attempting to withdraw up to the over allocation based on weight, and then unbonding additionally pro rata based on relative funds remaining. This pro rata portion is what could potentially round down, so funds can never be undelegated beyond what is staked.

Cantina Managed: Fix verified.

3.4.3 Incorrect withdrawal delay

Severity: Low Risk

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

Description: In `Staker.getWithdrawDelay`, we hardcode the return value for Monad Mainnet as 7:

```
function getWithdrawDelay() internal view returns (uint8) {
        // Monad Mainnet
        if (block.chainid == 143) return 7;
```

However, according to the most recent documentation, the `WITHDRAWAL_DELAY` is listed as 1.

Recommendation: Adjust the return value for Monad Mainnet to 1:

```
  function getWithdrawDelay() internal view returns (uint8) {
          // Monad Mainnet
-          if (block.chainid == 143) return 7;
+          if (block.chainid == 143) return 1;
```

Kintsu: Fixed in commit c8e58cf.

Cantina Managed: Fixed as recommended.