Telcoin Contracts
Security Review

Cantina Managed review by:

R0bert, Lead Security Researcher
Kaden, Security Researcher

June 26, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	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

Telcoin creates low-cost, high-quality financial products for every mobile phone user in the world.

From Jun 2nd to Jun 5th the Cantina team conducted a review of telcoin-laboratories-contracts on commit hash 5d6a1a64. The team identified a total of **11** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	4	4	0
High Risk	1	0	1
Medium Risk	2	2	0
Low Risk	2	2	0
Gas Optimizations	1	1	0
Informational	1	1	0
Total	**11**	**10**	**1**

3 Findings

3.1 Critical Risk

3.1.1 Incorrect double-negation of liquidityDelta inflates stored liquidity

Severity: Critical Risk

Context: TELxIncentiveHook.sol#L118

Description: `TELxIncentiveHook._beforeRemoveLiquidity` is invoked by the Uniswap V4 `PoolManager` contract with a negative `params.liquidityDelta` whenever an LP removes liquidity. Instead of forwarding this signed value as-is, the hook negates it once more:

```
registry.addOrUpdatePosition(
    sender,
    key.toId(),
    params.tickLower,
    params.tickUpper,
    -int128(params.liquidityDelta) // <<<
);
```

Because the value is now positive, `PositionRegistry.addOrUpdatePosition` executes the "add liquidity" branch:

```
pos.liquidity += uint128(liquidityDelta); // treated as a deposit
```

Therefore:

- The registry's `pos.liquidity` can only increase; it never decreases or reaches zero.
- Closed positions remain in `activePositionIds`, bloating storage and gas.
- Off-chain reward logic and governance weight calculations are severely inflated, enabling fraudulent reward claims and distorted voting power.

Any malicious user can abuse this by constantly adding and removing liquidity in the same tick range to inflate its `pos.liquidity` amount.

Recommendation: Consider forwarding the liquidity delta without altering its sign:

```
registry.addOrUpdatePosition(
    provider,                    // resolve real LP address if routed
    key.toId(),
    params.tickLower,
    params.tickUpper,
    int128(params.liquidityDelta)    // keep original sign
);
```

Telcoin: Fixed in PR 30.

Cantina Managed: Fix verified.

3.1.2 Hook trusts sender parameter directly as LP/Trader

Severity: Critical Risk

Context: TELxIncentiveHook.sol#L90, TELxIncentiveHook.sol#L114, TELxIncentiveHook.sol#L145

Description: Every Uniswap V4 hook callback supplies a `sender` parameter. According to the official Uniswap V4 docs, this value is the contract that called `PoolManager` (typically a swap or liquidity router), not the end-user's EOA or smart account. In `TELxIncentiveHook` all three implemented callbacks treat `sender` as if it were the actual liquidity provider or trader:

```
// beforeAddLiquidity & beforeRemoveLiquidity
registry.addOrUpdatePosition(
    sender,           // ← assumed to be the LP, but is usually a router
    key.toId(),
    // ...
);

// afterSwap
emit SwapOccurredWithTick(
    key.toId(),
    sender,           // ← assumed to be the swap initiator
    // ...
);
```

Because of this positions are recorded under router addresses rather than the real LPs. When an LP later calls `claim`, `unclaimedRewards[msg.sender]` is zero as the rewards accrue to the router contract and may be permanently inaccessible. Moreover, off-chain systems that rely on the emitted `SwapOccurredWithTick` event will log the router as the trader, masking the true flow of funds.

Recommendation:

- Maintain a whitelist of trusted routers in the hook.
- If sender is in the whitelist, call its public `msgSender()` view function to obtain the original EOA/smart-account; otherwise treat sender as the user (direct call flow).

For example:

```
function _resolveProvider(address maybeRouter) internal view returns (address) {
    if (trustedRouters[maybeRouter]) {
        return IRouter(maybeRouter).msgSender();  // must revert if not implemented
    }
    return maybeRouter; // direct interaction
}
```

Telcoin: Fixed in PR 32.

Cantina Managed: Fix verified.

3.1.3 Missing pool whitelist functionality in `TELxIncentiveHook`

Severity: Critical Risk

Context: UniswapAdaptor.sol#L31, VotingWeightCalculator.sol#L74

Description: `TELxIncentiveHook` has no mechanism to verify that a liquidity event belongs to a legitimate, TEL-denominated pool. Any address can deploy a Uniswap-V4 pool with two worthless ERC-20s, set `TELxIncentiveHook` as the pool's hook and call `modifyLiquidity` at will. Because the hook treats any pool as a legit TEL pool, the attacker's bogus pool creates real `PositionRegistry` entries. Later, `UniswapAdaptor.balanceOf` treats those positions as valid voting weight:

```
IPositionRegistry.Position[] memory positions = registry.getAllActivePositions();
for (uint256 i = 0; i < positions.length; i++) {
    if (positions[i].provider != voter) continue;
    // ...
    // price & amounts computed-even for non-TEL pools
    totalVotingWeight += amount0 + amount1InTEL;
}
```

The main impacts of this issue are:

- Unlimited voting-power minting: the attacker mints arbitrary "liquidity" units in a valueless pool and receives TEL-equivalent voting weight without depositing a single TEL.
- Denial-of-service risk: mass-creating pools inflates `activePositionIds`, causing gas-heavy loops in both `PositionRegistry` and `UniswapAdaptor` to exceed block gas limits and DoS legitimate users.

Recommendation: Maintain an on-chain mapping `allowedPoolId → true` inside the hook or registry and reject `addOrUpdatePosition` calls whose `poolId` is not explicitly authorized by governance. Similarly, in the `_afterSwap` function, do not emit an event unless the `poolId` is whitelisted.

Alternatively, implement a `beforeInitialize` hook which reverts initialization for untrusted pools, either with access control by validating the `sender` parameter to be a trusted account, or by maintaining an allowlist of valid `PoolKey`s.

Telcoin: Fixed in PR 29.

Cantina Managed: Fix verified.

3.1.4 Price-scaling error and superfluous 1e18 factor skew voting-weight calculation

Severity: Critical Risk

Context: UniswapAdaptor.sol#L62-L66, PositionRegistry.sol#L131-L135

Description: `getVotingWeightInTEL` (and the identical logic inside `UniswapAdaptor`) attempts to convert the token-1 side of an LP position into TEL units:

```
// Intended: price = sqrtPriceX96² / 2^192  (plain ratio)
uint256 priceX96 = (uint256(sqrtPriceX96) * uint256(sqrtPriceX96)) >> 96;
// Convert amount1 to TEL
uint256 amount1InTEL = (amount1 * 1e18) / priceX96;
return amount0 + amount1InTEL;
```

Two problems arise in this specific code:

- Right-shifting by 96 removes only one Q64.96 scale factor. The true plain price requires dividing by 2^{192}, or, if you want to keep the value in Q64.96, you must retain the extra factor and cancel it later. As a result, `priceX96` is $2^{96} \approx 7.9 \times 10^{28}$ times too large, making `amount1InTEL` virtually zero.
- `amount0` and `amount1` are already denominated in wei (18-decimal units). Multiplying by `1e18` is not necessary.

Combined, these mistakes under-estimate voting power for positions dominated by token-1.

Recommendation: Keep the price in Q64.96 and cancel the scale in the numerator; let `FullMath.mulDiv` handle 512-bit precision to avoid overflow:

```
// 1. price in Q64.96 (still × 2^96)
uint256 priceX96 = FullMath.mulDiv(
    uint256(sqrtPriceX96),
    uint256(sqrtPriceX96),
    2**96
);
// 2. Convert token-1 amount into TEL wei
uint256 amount1InTEL = FullMath.mulDiv(
    amount1,          // wei of token-1
    1 << 96,          // multiply by 2^96 to cancel the scale
    priceX96
);
// 3. Sum with native TEL balance
return amount0 + amount1InTEL;
```

Telcoin: Fixed in PR 20.

Cantina Managed: Fix verified.

3.2 High Risk

3.2.1 LP position transfers are not tracked

Severity: High Risk

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

Description: In Uniswap V4 users can make use of the PositionManager for creating liquidity positions. The `PositionManager`mints and manages ERC721 tokens associated with each position. These NFT tokens can be transferred, transferring the ownership of that position.

When this NFT is tranferred:

- The `PositionRegistry` entry keeps the old provider address.

- The new owner later calls `removeLiquidity` and `_beforeRemoveLiquidity` is triggered with `sender = newOwner` and a negative liquidityDelta.
- `addOrUpdatePosition` computes a fresh `positionId` keyed by `newOwner`, finds `pos.liquidity == 0`, and then executes. `pos.liquidity -= uint128(-liquidityDelta);` which reverts due to an underflow.
- The legitimate new owner is blocked from withdrawing or collecting fees.
- Governance weight remains credited to the old owner while the new owner has none.

Recommendation: Consider implementing the Subscriber pattern defined at Uniswap v4 subscriber docs. For example:

- Deploy a minimal subscriber contract (`TELxSubscriber`) that forwards the transfer notice to the `PositionRegistry`.
- In PositionRegistry add `transferPosition` which looks up the old `positionId`, copies the struct to a new ID keyed by `newOwner`, deletes the old entry and updates `activePositionIds` / auxiliary indices.
- Require, at LP onboarding time, that each position sets `TELxSubscriber` as its subscriber (or have the front-end do it implicitly).

Telcoin: Acknowledged. The fix requires the implementation of a new smart contract. A subsequent audit will be performed in the future on this new contract.

3.3 Medium Risk

3.3.1 Unchecked 256-bit multiplication of `sqrtPrice` values can overflow and revert

Severity: Medium Risk

Context: UniswapAdaptor.sol#L62-L63, UniswapAdaptor.sol#L83, UniswapAdaptor.sol#L89, PositionRegistry.sol#L131-L132, PositionRegistry.sol#L155, PositionRegistry.sol#L162

Description: Both `PositionRegistry.getVotingWeightInTEL` and the "all-token-0" branch inside `getAmountsForLiquidity` present in `UniswapAdaptor` and `PositionRegistry`, multiply two Q64.96 square-root prices (`sqrtPriceX96`, `sqrtRatioAX96`, `sqrtRatioBX96`) directly in 256-bit arithmetic:

```
// getVotingWeightInTEL
uint256 priceX96 = (uint256(sqrtPriceX96) * uint256(sqrtPriceX96)) >> 96;

// getAmountsForLiquidity  - token0 branch
amount0 = FullMath.mulDiv(
    uint256(liquidity) << FixedPoint96.RESOLUTION,
    sqrtPriceBX96 - sqrtPriceAX96,
    uint256(sqrtPriceAX96) * sqrtPriceBX96     // ← 256-bit multiply
);
```

Because each `sqrtPriceX96` term is bounded only by the protocol's limit ($\approx 2^{160} - 1$ at tick ± 887272), their product can require up to 320 bits. Solidity ≥ 0.8 reverts on any 256-bit overflow, so when a high-tick pool is queried:

- `getVotingWeightInTEL` reverts, blocking UI and off-chain indexers.
- `getAmountsForLiquidity` reverts, breaking internal weight calculations.

Recommendation: Never multiply two 160-bit roots directly in 256-bit space. Use `FullMath.mulDiv`, which facilitates multiplication and division that can have overflow of an intermediate value without any loss of precision:

- Builds the full 512-bit product (`prod0`, `prod1`) in assembly.
- Performs the division before returning a 256-bit result.
- Fix for `getVotingWeightInTEL`:

```
uint256 priceX96 = FullMath.mulDiv(sqrtPriceX96, sqrtPriceX96, 2**96); // still ×2^96
uint256 amount1InTEL = FullMath.mulDiv(amount1, 1 << 96, priceX96);    // cancel scale
```

- Fix for "all-token-0" branch in getAmountsForLiquidity: Adopt the two-step overflow-safe pattern used in Uniswap's `LiquidityAmounts` library:

```
uint256 intermediate = FullMath.mulDiv(
    liquidity,
    sqrtPriceBX96 - sqrtPriceAX96,
    sqrtPriceBX96
);
amount0 = FullMath.mulDiv(
    intermediate,
    1 << 96, // 2^96
    sqrtPriceAX96
);
```

Telcoin: Fixed in PR 34.

Cantina Managed: Fix verified.

3.3.2 Hard-coded "TEL is always token0" assumption

Severity: Medium Risk

Context: PositionRegistry.sol#L129

Description: `PositionRegistry.getVotingWeightInTEL` and `UniswapAdaptor._getAmountsForLiquidity` both assume TEL is always `token0` in the underlying Uniswap V4 pools. All TEL-equivalent math then treats amount0 as TEL and converts only the token-1 side:

```
uint256 amount1InTEL = ...;   // price = token1 / TEL
return amount0 + amount1InTEL; // amount0 presumed to be TEL
```

This is not accurate in practice as due to the Uniswap token-ordering rule, when a new V4 pool is created, the two token addresses are sorted lexicographically:

```
(token0, token1) = sort(tokenA, tokenB);

function sort(address tokenA, address tokenB) internal pure returns (address token0, address token1) {
    require(tokenA != tokenB, "Identical addresses");
    (token0, token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
    require(token0 != address(0), "Zero address");
}
```

Developers cannot choose the order manually and therefore if TEL's address happens to be greater than the counter-asset's address (e.g., TEL/USDC where USDC address is lower), TEL becomes the `token1` in the Uniswap V4 pool. amount0 is then the non-TEL asset, yet it is counted as TEL, mispricing the position.

Recommendation: Check if the TEL token is the `token0` or `token1` dynamically, for example:

```
if (telIsToken0) {
    // amount0 is TEL
    telEquivalent = amount0 + FullMath.mulDiv(amount1, 1, price); // token1 → TEL
} else {
    // amount1 is TEL
    telEquivalent = amount1 + FullMath.mulDiv(amount0, price, 1); // token0 → TEL
}
```

Telcoin: Fixed in PR 35.

Cantina Managed: Fix verified.

3.4 Low Risk

3.4.1 Block gas limit can be reached due to unbounded number of LP Positions

Severity: Low Risk

Context: UniswapAdaptor.sol#L31-L37

Description: The `UniswapAdaptor.balanceOf(address)` function estimates a voter's TEL-denominated voting power by scanning every active LP position stored in `PositionRegistry`:

```
IPositionRegistry.Position[] memory positions = registry.getAllActivePositions();

for (uint256 i = 0; i < positions.length; i++) {
    if (positions[i].provider != voter) continue;
    // ...
}
```

Neither the registry nor the hook enforces a cap on:

- Global number of positions (`activePositionIds.length`).
- Per-user positions.

Therefore an attacker can create tens of thousands of tiny liquidity ranges (or use the fake-pool vector described earlier). Each `balanceOf` call then performs thousands of Uniswap math operations, easily exceeding the block gas limit. Another possibility, is a user unaware of this issue, accidentally opening too many positions. Note that this risk also impacts the removal case of `PositionRegistry.addOrUpdatePosition` since we must loop through (up to) all `activePositionIds`:

```
if (pos.liquidity == 0) {
    // ...
    for (uint256 i = 0; i < activePositionIds.length; i++) {
        if (activePositionIds[i] == positionId) {
            activePositionIds[i] = activePositionIds[
                activePositionIds.length - 1
            ];
            activePositionIds.pop();
            break;
        }
    }
    // ...
}
```

Recommendation: Since we only access one `voter` at a time in `UniswapAdaptor.balanceOf`, we can replace the `activePositionsIds` array with a per-`provider` mapping which points to all of the given `providers` active positions, e.g., `providerPositions[provider]`. This way, instead of using `registry.getAllActivePositions` and looping through all positions regardless of the `provider`/`voter`, we can just retrieve the positions of the given `voter`. This would significantly reduce the risk of a block gas limit DoS while also significantly reducing gas consumption.

Consider also implementing a maximum amount of positions which a `provider` can have open at a time to completely eliminate the risk of DoS. This can be accomplished by updating the `PositionRegistry.addOrUpdatePosition` function to revert `if (providerPositions[provider].length > MAX_POSITIONS)` (e.g., 100).

Telcoin: Fixed in PR 37 and PR 38.

Cantina Managed: Fix verified.

3.4.2 `PositionUpdated` event not emitted when liquidity is removed

Severity: Low Risk

Context: PositionRegistry.sol#L215-L248

Description: In `PositionRegistry.addOrUpdatePosition`, when liquidity is added, we emit the `PositionUpdated` event and when liquidity is entirely removed, we emit the `PositionRemoved` event:

```
if (liquidityDelta > 0) {
    // ...
    emit PositionUpdated(
        positionId,
        provider,
        poolId,
        tickLower,
        tickUpper,
        pos.liquidity
    );
} else {
    // ...

    if (pos.liquidity == 0) {
        // ...
        emit PositionRemoved(
            positionId,
            provider,
            poolId,
            tickLower,
            tickUpper
        );
    }
}
```

However, in case only some of the available liquidity is removed, we don't emit the `PositionUpdated` event, even though the position is in fact being updated. As a result, liquidity positions will not be correctly tracked by event indexers in this case.

Recommendation: Emit a `PositionUpdated` event in the liquidity removal case when liquidity is not fully removed:

```
  if (liquidityDelta > 0) {
      // ...
      emit PositionUpdated(
          positionId,
          provider,
          poolId,
          tickLower,
          tickUpper,
          pos.liquidity
      );
  } else {
      // ...

      if (pos.liquidity == 0) {
          // ...
          emit PositionRemoved(
              positionId,
              provider,
              poolId,
              tickLower,
              tickUpper
          );
-     }
+     } else {
+         emit PositionUpdated(
+             positionId,
+             provider,
+             poolId,
+             tickLower,
+             tickUpper,
+             pos.liquidity
+         );
+     }
  }
```

Telcoin: Fixed in PR 18.

Cantina Managed: Fix verified.

3.5 Gas Optimization

3.5.1 Repeated condition

Severity: Gas Optimization

Context: PositionRegistry.sol#L207-L225

Description: In `PositionRegistry.addOrUpdatePosition`, we check if liquidity is being added, and if there is not yet any liquidity, we initialize the position. Afterwards, we again check whether liquidity is being added and execute relevant logic:

```
if (pos.liquidity == 0 && liquidityDelta > 0) {
    pos.provider = provider;
    pos.poolId = poolId;
    pos.tickLower = tickLower;
    pos.tickUpper = tickUpper;
    activePositionIds.push(positionId);
}

if (liquidityDelta > 0) {
    pos.liquidity += uint128(liquidityDelta);
    emit PositionUpdated(
        positionId,
        provider,
        poolId,
        tickLower,
        tickUpper,
        pos.liquidity
    );
} else {
    // ...
}
```

We can optimize this, checking `liquidityDelta > 0` once, by nesting the first `if` statement within the second `if` statement (see Recommendation).

Recommendation: Nest the first `if` statement within the second one, removing the repeated `liquidityDelta > 0` condition:

```
// If adding liquidity
if (liquidityDelta > 0) {
   // If there isn't yet any liquidity in the position
   if (pos.liquidity == 0) {
        // Initialize the position
        pos.provider = provider;
        pos.poolId = poolId;
        pos.tickLower = tickLower;
        pos.tickUpper = tickUpper;
        activePositionIds.push(positionId);
    }

    pos.liquidity += uint128(liquidityDelta);
    emit PositionUpdated(
        positionId,
        provider,
        poolId,
        tickLower,
        tickUpper,
        pos.liquidity
    );
} else {
    // ...
}
```

3.6 Informational

3.6.1 Same liquidity can be re-used across multiple wallets to inflate voting power

Severity: Informational

Context: PositionRegistry.sol#L109-L118

Description: `PositionRegistry.getVotingWeightInTEL()` (and its wrapper in `UniswapAdaptor`) returns the live TEL-equivalent value of an LP position. If this function is consumed by a governance module without an historical-balance snapshot, the same liquidity can vote many times in one proposal cycle.

This intention is suggested in the documentation provided related to the `VotingWeightCalculator` contract which states that the `VotingWeightCalculator` contract aggregates the voting power of an address by querying a list of external contracts (called "sources") that implement the `ISource` interface. This modular structure allows the voting system to evolve and include new liquidity mechanisms or staking sources without redeploying the core logic.

- Modular Voting Power Calculation: Delegates responsibility for calculating `balanceOf(address)` to external contracts. conforming to `ISource`. This enables flexible support for different types of TEL token holdings (e.g. LP positions, staked tokens, etc).
- Dynamic Source Management: By storing voting logic in modular source contracts, the system avoids. centralizing all logic in one place and allows on-chain upgrades by adding/removing sources.

Example Flow:

1. DAO adds a new liquidity contract by calling `addSource(<address>)`.
2. A frontend or off-chain voting engine calls `balanceOf(user)` on `VotingWeightCalculator`.
3. The calculator queries each source contract to sum the user's balance and returns the aggregate voting power.

In order to exploit this:

- Wallet A adds liquidity, calls `balanceOf`, votes, then removes liquidity.
- Wallet B adds the very same liquidity (or receives it via `transferPosition`), calls `balanceOf`, votes, removes.
- The process can be repeated for N wallets.

Because nothing in the contracts allows recording the balances at `proposalCreationBlock - 1`, each wallet's transient liquidity is accepted at face value, letting an attacker cast arbitrarily many votes with a single TEL stake.

Recommendation: Consider adopting a block-snapshot pattern:

- At proposal creation, store `snapshotPower[voter] = balanceOf(voter)` using the previous block number.
- During `castVote`, read the cached value instead of re-calling `balanceOf`.

A snapshot prevents liquidity that has already influenced a proposal from being counted again, eliminating the multi-wallet vote-inflation vector.

Telcoin: Fixed as the issue is mitigated by the Snapshot platform itself. We acknowledge that, without a robust system in place to address this vulnerability, voting would be susceptible to compromise and therefore unsuitable for secure use.

Cantina Managed: Fix verified.