Telcoin Application Network Improvement Proposal 1 Contracts

Security Review

Cantina Managed review by:

Phaze, Lead Security Researcher
Philogy, Security Researcher

March 10, 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 Mar 4th to Mar 7th the Cantina team conducted a review of telcoin-monorepo on commit hash c65d8bb3. The team identified a total of **8** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	2	2	0
Gas Optimizations	1	1	0
Informational	4	3	1
Total	**8**	**7**	**1**

3 Findings

3.1 Medium Risk

3.1.1 `_incrementCumulativeRewards` incorrectly updates cumulative rewards

Severity: Medium Risk

Context: TANIssuanceHistory.sol#L177

Summary: The `_incrementCumulativeRewards` method updates using `block.number` when it should use `endBlock` as `increaseClaimableByBatch` updates are intended to be posted 64/500 blocks after the period once the on-chain data is finalized meaning recording the submission block number as the block for the update would be incorrect.

Impact is medium as the cumulative rewards tracker will be relied upon to accurately report staker rewards for calculations off-chain.

Likelihood Explanation: The likelihood is **high** because it is intended that this method will be called on a regular basis to push updates on chain. Furthermore the docs describing how the rewards are calculated states (under tn-contracts/src/issuance/README.md "Note on Polygon chain reorgs"):

> [...] the offchain calculator refuses to process or produce calldata for an `endBlock` which is less than 500 blocks earlier than the current block. [...].

This means we can expect that `endBlock != block.number` every time that `increaseClaimableByBatch` is called.

Impact Explanation: The impact is **low** because none of the in-scope contracts rely on the integrity of the `_cumulativeRewards` store & furthermore the client has stated that *for now* the offchain calculator also does not rely on the contract's `_cumulativeRewards` store for its calculations.

Recommendation: Ensure that when `_cumulativeRewards` is updated the block number that's used is the `endBlock` and not the current block number to ensure that the data stored on-chain is consistent:

```
@@ -104,7 +104,7 @@ contract TANIssuanceHistory is Ownable {
     function increaseClaimableByBatch(
         address[] calldata accounts,
         uint256[] calldata amounts,
-        uint256 endBlock
+        uint32 endBlock
     )
         external
         onlyOwner
@@ -121,7 +121,7 @@ contract TANIssuanceHistory is Ownable {
             if (amounts[i] == 0) continue;

             totalAmount += amounts[i];
-            _incrementCumulativeRewards(accounts[i], amounts[i]);
+            _incrementCumulativeRewards(accounts[i], amounts[i], endBlock);
         }

         // set approval as `SimplePlugin::increaseClaimableBy()` pulls TEL from this address to itself
@@ -170,11 +170,11 @@ contract TANIssuanceHistory is Ownable {
     /**
      * Internals
      */
-    function _incrementCumulativeRewards(address account, uint256 amount) internal {
+    function _incrementCumulativeRewards(address account, uint256 amount, uint32 endBlock) internal {
         uint256 prevCumulativeReward = cumulativeRewards(account);
         uint224 newCumulativeReward = SafeCast.toUint224(prevCumulativeReward + amount);

-        _cumulativeRewards[account].push(uint32(block.number), newCumulativeReward);
+        _cumulativeRewards[account].push(endBlock, newCumulativeReward);
     }

     /// @dev Validate that user-supplied block is in the past, and return it as a uint48.
```

Telcoin: Fixed in commit 1060bb3

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 Unchecked return value from `increaseClaimableBy()` function call

Severity: Low Risk

Context: TANIssuanceHistory.sol#L131

Description: In the TANIssuanceHistory contract, the `increaseClaimableByBatch()` function calls `tanIssuancePlugin.increaseClaimableBy(accounts[i], amounts[i])` but does not check the return value. The `increaseClaimableBy()` function in the SimplePlugin contract returns a boolean value, but it's ambiguous whether this value indicates a success status or simply whether the amount is non-zero.

Currently, the SimplePlugin implementation returns `false` only when the reward amount is zero, which is already checked in the TANIssuanceHistory contract. However, if the plugin is upgraded in the future to return `false` for other failure conditions, the lack of return value checking could allow accounting discrepancies to develop without any error indication.

Recommendation: If the return value is meant to represent an operational success status, consider adding a check for the return value from `increaseClaimableBy()` and handle failures appropriately:

```
+ error IncreaseClaimableFailed(address account, uint256 amount);

  function increaseClaimableByBatch(
      address[] calldata accounts,
      uint256[] calldata amounts,
      uint256 endBlock
  )
      external
      onlyOwner
      whenNotDeactivated
  {
      // ... existing code ...

      tel.approve(address(tanIssuancePlugin), totalAmount);
      for (uint256 i; i < len; ++i) {
          if (amounts[i] == 0) continue;
-         tanIssuancePlugin.increaseClaimableBy(accounts[i], amounts[i]);
+         bool success = tanIssuancePlugin.increaseClaimableBy(accounts[i], amounts[i]);
+         if (!success) revert IncreaseClaimableFailed(accounts[i], amounts[i]);
      }
  }
```

Telcoin: As currently structured, documented in its NatSpec, and noted by the finding, the SimplePlugin allows only for two branches: true for a successful transfer of nonzero token amount and false for an attempted transfer of zero token amount. Any other scenario results in a revert because the TEL token returns false for failures which would revert `SafeERC20::safeTransferFrom()`.

We agree that this choice of forked code paths is unusual and arguably not very helpful for external contracts which use the SimplePlugin as a dependency. It is a choice contributing more to tech debt than anything else, however this API is now relied on by multiple other Telcoin codebases and deployments.

For now, we will use the recommendation which keeps this design decision and handles it at the dependency level (i.e. the `TANIssuanceHistory` contract's side) by adding a check ensuring `SimplePlugin::increaseClaimableBy()` returns true. As noted by the finding, this sets an expectation for future upgrades and contributes a clear 'best practice' for building on top of this version of the SimplePlugin. The recommendation also pairs well with the first gas optimization recommendation by introducing a revert path for poorly constructed calldata.

Side note: *While at this time making a change to the SimplePlugin's expected behavior would incur more dev time/security considerations in other codebases than is worth for improving this code branching tech debt, we have plans for larger changes to the `SimplePlugin` (and its `StakingModule`) slated for a migration from Polygon to Telcoin Network at its mainnet launch. That will be a more appropriate time to address the underlying design choice.*

Resolved in commit a8100d3.

Cantina Managed: Fix verified.

3.2.2 Misleading claimable balances after plugin deactivation

Severity: Low Risk

Context: SimplePlugin.sol#L82-L85

Description: The SimplePlugin contract doesn't account for its deactivated state when reporting claimable balances through `claimable()` and `totalClaimable()` functions. When a plugin is deactivated, users can no longer claim rewards, but these functions continue to report the full reward balances as if they were still claimable.

This creates a discrepancy between what the user interface may show as available rewards and what users can actually claim. The StakingModule relies on these functions to calculate total balances, which means it will report incorrect values for user balances and the total supply after plugin deactivation. This could lead to confusion for users who see rewards displayed that they cannot actually claim.

Recommendation: Modify the `claimable()` and `totalClaimable()` functions in SimplePlugin to return 0 when the plugin is deactivated:

```
  function claimable(
      address account,
      bytes calldata
  ) external view override returns (uint256) {
+     if (deactivated()) {
+         return 0;
+     }
      return _claimable[account].latest();
  }

  function totalClaimable() external view override returns (uint256) {
+     if (deactivated()) {
+         return 0;
+     }
      return _totalOwed;
  }
```

Alternatively, consider creating a distinction between "owed" amounts (the total that would be owed if the plugin wasn't deactivated) and "claimable" amounts (what can actually be claimed now):

```
// Return what can actually be claimed now
function claimable(
    address account,
    bytes calldata
) external view override returns (uint256) {
    if (deactivated()) {
        return 0;
    }
    return _claimable[account].latest();
}

// Return what is tracked/owed regardless of deactivation
function totalOwed() external view returns (uint256) {
    return _totalOwed;
}
```

This approach would provide better clarity after a plugin has been deactivated and would prevent misleading balance displays in user interfaces.

Telcoin: Great nuance considering the contract's downstream implications across the full stack, through frontend. Added recommended checks for `deactivated()` status in `claimable()` and `totalClaimable()` functions, as well as delineated the distinction between "claimable" and "owed" nomenclature by exposing two new external functions `owed()` and `totalOwed()` which replicate the previous (deactivation agnostic) behavior of `claimable()` and `totalClaimable()`, respectively.

Resolved in commit 819a7af.

Cantina Managed: Fix verified.

3.3 Gas Optimization

3.3.1 Gas optimization recommendations

Severity: Gas Optimization

Context: TANIssuanceHistory.sol#L131

Description and Recommendations:

1. Skip Zero Amount Transfers. In `increaseClaimableByBatch()`, there's inconsistent handling of zero amounts. Zero checks are performed before updating local storage but not before calling the plugin:

```
   function increaseClaimableByBatch(
       address[] calldata accounts,
       uint256[] calldata amounts,
       uint256 endBlock
   ) external onlyOwner whenNotDeactivated {
       // ...
       for (uint256 i; i < len; ++i) {
           if (amounts[i] == 0) continue;

           totalAmount += amounts[i];
           _incrementCumulativeRewards(accounts[i], amounts[i]);
       }

       tel.approve(address(tanIssuancePlugin), totalAmount);
       for (uint256 i; i < len; ++i) {
+          if (amounts[i] == 0) continue;
           tanIssuancePlugin.increaseClaimableBy(accounts[i], amounts[i]);
       }
   }
```

   Alternatively, remove zero amount checks, as the code should not be optimized for unintended use cases.

2. Cache Frequently Used Storage Variables: The `tanIssuancePlugin` is a storage variable meaning that every time it's referenced it'll cause a new storage read. Cache it in a local variable to avoid redundant storage access:

```
   function increaseClaimableByBatch(
       address[] calldata accounts,
       uint256[] calldata amounts,
       uint256 endBlock
   ) external onlyOwner whenNotDeactivated {
       uint256 len = accounts.length;
       if (amounts.length != len) revert ArityMismatch();

       if (endBlock > block.number) revert InvalidBlock(endBlock);
       lastSettlementBlock = endBlock;

+      ISimplePlugin plugin = tanIssuancePlugin;
       uint256 totalAmount;
       for (uint256 i; i < len; ++i) {
           if (amounts[i] == 0) continue;

           totalAmount += amounts[i];
           _incrementCumulativeRewards(accounts[i], amounts[i]);
       }

-      tel.approve(address(tanIssuancePlugin), totalAmount);
+      tel.approve(address(plugin), totalAmount);
       for (uint256 i; i < len; ++i) {
           if (amounts[i] == 0) continue;
-          tanIssuancePlugin.increaseClaimableBy(accounts[i], amounts[i]);
+          plugin.increaseClaimableBy(accounts[i], amounts[i]);
       }
   }
```

   Small caveat: If `increaseClaimableBy` is able to reenter and somehow modify `tanIssuancePlugin` this change will result in different functionality.

3. Redundant Type Casting in `_validateQueryBlock()`:** Multiple safecasts can be simplified by immediately casting to the required type:

```
- function _validateQueryBlock(uint256 queryBlock) internal view returns (uint48) {
+ function _validateQueryBlock(uint256 queryBlock) internal view returns (uint32) {
    uint48 currentBlock = clock();
    if (queryBlock > currentBlock) revert FutureLookup(queryBlock, currentBlock);
-     return SafeCast.toUint48(queryBlock);
+     return SafeCast.toUint32(queryBlock);
}

- function _cumulativeRewardsAtBlock(address account, uint48 queryBlock) internal view returns (uint256)
↪ {
+ function _cumulativeRewardsAtBlock(address account, uint32 queryBlock) internal view returns (uint256)
↪ {
-     return _cumulativeRewards[account].upperLookupRecent(SafeCast.toUint32(queryBlock));
+     return _cumulativeRewards[account].upperLookupRecent(queryBlock);
}
```

4. `SimplePlugin.increaseClaimableBy` not optimized for token transfer & batched calls: The current approach to transferring funds from the `TANIssuanceHistory` contract to the `SimplePlugin` contract is sub optimal because it requires a call to `approve` and several calls to `transferFrom` in `SimplePlugin`.

   Saving a lot of gas, if accommodated for it `SimplePlugin` could receive TEL tokens from the issuance contract via a simple transfer by leveraging its existing `_totalOwed` variable.

   Mainly it could simply check that `tel.balanceOf(address(this)) >= _totalOwed + amount` for every call or even better only once if it were to process the `increaseClaimableBy` updates as a batch. `ERC20.transfer` is much cheaper than approve + `transferFrom` because those methods need to update an additional allowance state variable vs. `transfer` which just directly updates balances.

   Non batched example:

```
  function increaseClaimableBy(address account, uint256 amount)
      external
      whenNotDeactivated
      onlyIncreaser
      returns (bool)
  {
      // if amount is zero do nothing
      if (amount == 0) {
          return false;
      }

+     // Assume tokens have already been transferred and check
+     uint256 newOwed = _totalOwed + amount;
+     require(newOwed <= tel.balanceOf(address(this)), "Amount > transferred funds");


      // keep track of old claimable and new claimable
      uint256 oldClaimable = _claimable[account].latest();
      uint256 newClaimable = oldClaimable + amount;

      // update _claimable[account] with newClaimable
      _claimable[account].push(newClaimable);

      // update _totalOwed
-     _totalOwed += amount;
+     _totalOwed = newOwed;

-     // transfer TEL
-     tel.safeTransferFrom(msg.sender, address(this), amount);
-
      emit ClaimableIncreased(account, oldClaimable, newClaimable);

      return true;
  }
```

5. Redundant `whenNotDeactivated` modifier in `TANIssuanceHistory`: (context: TANIssuanceHistory.sol#L111). The `whenNotDeactivated` modifier is redundant because the method is already `onlyOwner` guarded and also the inner `tanIssuancePlugin.increaseClaimableBy` calls also complete that check.

6. Replace pairs of same-length arrays with single array of tuples: By redefining the pair of `address[] calldata accounts` & `uint256[] calldata amounts` arrays as a single array of structs (recognized as

tuples in the ABI) with an `address account` and `uint256 amount` field in the `increaseClaimableBy-Batch` method, you make it more efficient by removing the need for the runtime length check as well as making the calldata footprint smaller. It also allows the loop over elements to be more efficient as you only need to do index bounds checking once per iteration.

Example implementation:

```
+  struct Increase {
+      address account;
+      uint256 amount;
+  }
+
   /// @dev Saves the settlement block, updates cumulative rewards history, and settles TEL rewards on
   ↪  the plugin
   function increaseClaimableByBatch(
-      address[] calldata accounts,
-      uint256[] calldata amounts,
+      Increase[] calldata increases,
       uint256 endBlock
   )
       external
       onlyOwner
       whenNotDeactivated
   {
-      uint256 len = accounts.length;
-      if (amounts.length != len) revert ArityMismatch();
+      uint256 len = increases.length;

       if (endBlock > block.number) revert InvalidBlock(endBlock);
       lastSettlementBlock = endBlock;

       uint256 totalAmount;
       for (uint256 i; i < len; ++i) {
-          if (amounts[i] == 0) continue;
-
-          totalAmount += amounts[i];
-          _incrementCumulativeRewards(accounts[i], amounts[i]);
+          Increase calldata increase = increases[i];
+          totalAmount += increase.amount;
+          _incrementCumulativeRewards(increase.account, increase.amount);
       }

       // set approval as `SimplePlugin::increaseClaimableBy()` pulls TEL from this address to itself
       tel.approve(address(tanIssuancePlugin), totalAmount);
       for (uint256 i; i < len; ++i) {
+          Increase calldata increase = increases[i];
           // event emission on this contract is omitted since the plugin emits a `ClaimableIncreased`
           ↪  event
-          tanIssuancePlugin.increaseClaimableBy(accounts[i], amounts[i]);
+          tanIssuancePlugin.increaseClaimableBy(increase.account, increase.amount);
       }
   }
```

7. Consider Off-Chain Tracking for Historical Rewards: Given that reward computations already rely heavily on off-chain calculations, it might not be necessary to store explicit checkpoints of cumulative rewards on-chain. This data could be computed off-chain through indexing/tracking events, similar to how the entire reward calculation is currently handled.

```
- mapping(address => Checkpoints.Trace224) private _cumulativeRewards;

  function increaseClaimableByBatch(
      address[] calldata accounts,
      uint256[] calldata amounts,
      uint256 endBlock
  ) external onlyOwner whenNotDeactivated {
      // ...
      for (uint256 i; i < len; ++i) {
          if (amounts[i] == 0) continue;

          totalAmount += amounts[i];
-         _incrementCumulativeRewards(accounts[i], amounts[i]);
+         emit RewardIncreased(accounts[i], amounts[i], endBlock);
      }
      // ...
  }
```

This approach would require additional off-chain infrastructure to track and serve historical reward data, but could substantially reduce on-chain costs while maintaining the same trust model.

Telcoin: These optimizations increase the maximum number of accounts we can reward before hitting out of gas errors, so we would be able to handle more eligible TANIP-1 program participants.

1. Fixed in commit e9fc684.
2. Fixed in commit 594d3c1.
3. This does not affect gas usage as it is only used in read-only view contexts however the readability improvements are definitely worthwhile. Fixed in commit 4bbec8d.
4. Acknowledged to be more gas efficient than the existing implementation for `SimplePlugin::increaseClaimableBy()` when combined with an `ERC20::transfer()` on the `TANIssuanceHistory`'s side (rather than the current `ERC20::approve()` & `ERC20::transferFrom()`). However such a change to the SimplePlugin implementation would break its current API and expected behavior, which is relied upon by multiple other components and programs in Telcoin infrastructure. In this case, the gas savings are outweighed by the time required to adjust for broken backwards compatibility in other codebases/deployments which rely on the SimplePlugin's `approve` and `transferFrom` flow as is.
5. Fixed in commit e414b13.
6. Fixed in commit 2f7c34.

Cantina Managed: Fixes verified.

3.4 Informational

3.4.1 Consider explicit validation of reward settlement chronology

Severity: Informational

Context: TANIssuanceHistory.sol#L116-L117

Summary: The TANIssuanceHistory contract's `increaseClaimableByBatch()` function could benefit from explicit chronological validation of reward settlements to improve clarity.

Description: The `increaseClaimableByBatch()` function processes rewards for multiple accounts and stores historical checkpoints. While the OpenZeppelin Checkpoints library inherently prevents non-monotonic updates to the checkpoints themselves, the contract does not explicitly verify that each new `endBlock` parameter is greater than or equal to the previous `lastSettlementBlock`.

This implicit protection works well for the checkpoint data, but having an explicit check for the `lastSettlementBlock` variable would enhance code readability and help prevent potential misunderstandings during administrative operations.

Note that there is a related issue where the `endBlock` parameter is not being correctly passed to the `_incrementCumulativeRewards()` function, which is addressed separately.

Recommendation: Consider adding an explicit check to validate chronological ordering or adding documentation to clarify the behavior:

```
  function increaseClaimableByBatch(
      address[] calldata accounts,
      uint256[] calldata amounts,
      uint256 endBlock
  )
      external
      onlyOwner
      whenNotDeactivated
  {
      uint256 len = accounts.length;
      if (amounts.length != len) revert ArityMismatch();
      if (endBlock > block.number) revert InvalidBlock(endBlock);
+     // Consider adding validation to ensure chronological processing
+     // if (endBlock < lastSettlementBlock) revert InvalidBlock(endBlock);

+     // Note: While not explicitly checked above, the Checkpoints library
+     // inherently prevents non-monotonic updates to historical data
      lastSettlementBlock = endBlock;
      uint256 totalAmount;
      for (uint256 i; i < len; ++i) {
          if (amounts[i] == 0) continue;
          totalAmount += amounts[i];
          _incrementCumulativeRewards(accounts[i], amounts[i]);
      }

      // Rest of the function...
  }
```

Telcoin: Fixed in commit e62c390.

Cantina Managed: Fix verified.

3.4.2 Missing validation of token address when setting TAN issuance plugin

Severity: Informational

Context: TANIssuanceHistory.sol#L140

Description: In the TANIssuanceHistory contract, the `setTanIssuancePlugin()` function allows the owner to set a new issuance plugin. However, it does not verify that the new plugin's TEL token address matches the current TEL token address. This could lead to inconsistencies if a plugin with a different token address is set. The current implementation only checks that the new plugin address is not the zero address and that it contains code, but does not validate the compatibility of the token addresses:

```
function setTanIssuancePlugin(ISimplePlugin newPlugin) external onlyOwner {
    if (address(newPlugin) == address(0x0) || address(newPlugin).code.length == 0) {
        revert InvalidAddress(address(newPlugin));
    }
    tanIssuancePlugin = newPlugin;
}
```

Recommendation: Add a validation check to ensure that the new plugin's TEL token address matches the current TEL token address:

```
  error IncompatiblePlugin();

  function setTanIssuancePlugin(ISimplePlugin newPlugin) external onlyOwner {
-     if (address(newPlugin) == address(0x0) || address(newPlugin).code.length == 0) {
-         revert InvalidAddress(address(newPlugin));
-     }
+     // Ensure the new plugin uses the same TEL token
+     if (newPlugin.tel() != tel) {
+         revert IncompatiblePlugin();
+     }
      tanIssuancePlugin = newPlugin;
  }
```

This additional check provides a defensive measure against configuration errors that could lead to operational issues and removes the need to check for the zero address or an address without code.

Telcoin: Fixed in commit d970896.

Cantina Managed: Fix verified.

3.4.3 Use `Ownable2Step` instead of `Ownable` for better security

Severity: Informational

Context: TANIssuanceHistory.sol#L21

Description: The TANIssuanceHistory contract inherits from OpenZeppelin's `Ownable` contract, which allows for immediate ownership transfers. This creates a risk where ownership could be accidentally transferred to an incorrect or inaccessible address, resulting in the contract becoming permanently locked without an owner.

The `Ownable2Step` pattern from OpenZeppelin provides a more secure ownership transfer mechanism by requiring the new owner to accept the transfer in a separate transaction. This two-step process reduces the risk of accidental transfers to incorrect addresses.

Recommendation: Replace the `Ownable` inheritance with `Ownable2Step`:

```
- import { Ownable } from "@openzeppelin/contracts/access/Ownable.sol";
+ import { Ownable2Step } from "@openzeppelin/contracts/access/Ownable2Step.sol";

- contract TANIssuanceHistory is Ownable {
+ contract TANIssuanceHistory is Ownable2Step {
```

This change maintains the same functionality while adding an additional security layer to ownership transfers. The new owner will need to call `acceptOwnership()` to complete the transfer process, preventing accidental transfers to incorrect addresses.

Telcoin: We acknowledge that Ownable2Step would improve security with its two transaction requirement however given our distribution flow forwards funds directly from the TAN governance safe to the SimplePlugin so as not to leave idle tokens in the history contract, the benefit is limited.

Owner permissions grants access to `increaseClaimableByBatch()`, `setTanIssuancePlugin()`, and `rescueTokens()`. The two former functions already require an explicit transaction granting the history contract `Increaser` permissions, which is already similar to a 2-step permissioning setup. Considering the distribution flow, the third function is already limited to tokens which are mistakenly sent to the history contract, an expectedly rare occurrence. The Telcoin team believes these tradeoffs are not worth the recommended change and that a high degree of confidence is better achieved by performing transaction simulation ahead of any onchain actions.

Cantina Managed: Acknowledged.

3.4.4 `SimplePlugin` implements `ERC165.supportsInterface` but is not ERC165 compliant

Severity: Informational

Context: SimplePlugin.sol#L104

Description: Per the specification of ERC165 contracts must return `true` when `supportsInterface` is called with the selector for `supportsInterface` (`0x01ffc9a7`) and `false` for `0xffffffff` however the plugin's implementation of `supportsInterface` only checks against `type(IPlugin).interfaceId` which would result in `false` when `supportsInterface(0x01ffc9a7)` is called.

Impact & Likelihood Explanation: The impact of this issue is that the standard check as recommended by the standard will fail. However due to ERC165's limited on-chain use combined with the fact that none of the in-scope contracts rely on conformance with the standard the likelihood & impact have both been evaluated to be low.

Recommendation: Update `supportsInterface` to also check for the standard method:

```
  function supportsInterface(bytes4 interfaceId) public view virtual override returns (bool) {
-     return interfaceId == type(IPlugin).interfaceId;
+     return interfaceId == type(IPlugin).interfaceId || interfaceId == this.supportsInterface.selector;
  }
```

Telcoin: Fixed in commit c7c02c45.

Cantina Managed: Fix verified.