CANTINA

Eco Routes

Security Review

Cantina Managed review by:
0xRajeev, Lead Security Researcher

0xWeiss, Security Researcher
Phaze, Security Researcher

January 3, 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

Eco enables apps to unlock stablecoin liquidity from any connected chain and give users the simplest onchain experience.

From Dec 4th to Dec 10th the Cantina team conducted a review of eco-routes on commit hash dbea76b6. A total of **22** issues were found.

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	2	2	0
High Risk	2	2	0
Medium Risk	1	0	1
Low Risk	8	5	3
Gas Optimizations	1	1	0
Informational	8	5	3
Total	**22**	**15**	**7**

3 Findings

3.1 Critical Risk

3.1.1 Anyone can arbitrarily set `provenStates[chainId].blockNumber` in `proveWorldStateCannon()` to prevent proving of all intents

Severity: Critical Risk

Context: Prover.sol#L449-L475

Description: Anyone can set `provenStates[chainId].blockNumber` to an arbitrarily large value, e.g. `type(uint256).max`, in `proveWorldStateCannon()` to thereafter prevent proving of all intents on destination `chainId` leading to loss of rewards for all solvers.

`proveWorldStateCannon()` allows solvers (or their set claimants) to validate world state on a source chain for Cannon-based Optimism L2 destination chains where they have fulfilled intents. This is a require step before they can call `proveIntent()` on a specific intent that they have fulfilled and are attempting to claim its rewards. `proveWorldStateCannon()` implements the following sequence of steps:

1. Validates that the `l1WorldStateRoot` argument is indeed the `existingSettlementBlockProof.stateRoot` for the `chainId` argument's settlement chain.
2. Validates that the `l2WorldStateRoot` argument corresponds to a resolved `DEFENDER_WINS` valid fault dispute game that has been generated by the L2's `settlementContract` on L1 rooted at `l1WorldStateRoot` from (1).
3. Encodes `rlpEncodedBlockData` argument into a `blockProof` to store the proved `l2WorldStateRoot` along with block number and hash (which are derived from `rlpEncodedBlockData`) only if `existingBlockProof.blockNumber < blockProof.blockNumber` so that only successive L2 blocks can be proved.

However, the `rlpEncodedBlockData` argument used in (3) is never validated to correspond to the `l2WorldStateRoot` argument validated in (2). This allows anyone to provide valid `l1WorldStateRoot` and `l2WorldStateRoot` in (1) and (2) but arbitrary `rlpEncodedBlockData` in (3) which gets used to encode `provenStates[chainId].blockNumber` and `provenStates[chainId].blockHash`.

The updated `provenStates[chainId].blockNumber` is used to determine if any successive `proveWorldStateCannon()` calls can update `provenStates[chainId] = blockProof` because of the `existingBlockProof.blockNumber < blockProof.blockNumber` check.

Any malicious participant can therefore provide `rlpEncodedBlockData` argument to `proveWorldStateCannon()` such that its elements corresponding to `blockNumber` equal to, for example, `type(uint256).max`. This will ensure that any successive `proveWorldStateCannon()` calls will revert with `OutdatedBlock` and therefore will never be able to prove any future L2 blocks.

Impact: High, because one can set `provenStates[chainId].blockNumber` to an arbitrarily large value, e.g. `type(uint256).max`, to thereafter prevent proving of all intents on destination `chainId` leading to loss of rewards for all solvers. One can do this for all supported Cannon-based chains to permanently stall proving and thereby halting the protocol on those chains because solvers will stop fulfilling intents there.

Likelihood: High, because this requires anyone to call `proveWorldStateCannon()` only once with valid L1+L2 state arguments (easily obtainable L1 and L2 world state roots) but with arbitrary `rlpEncodedBlockData`.

Recommendation: Consider validating `rlpEncodedBlockData` with a check such as:

```
require(keccak256(rlpEncodedBlockData) == disputeGameFactoryProofData.latestBlockHash, "Invalid block hash");
```

which constrains `rlpEncodedBlockData` to the previously proven `disputeGameFactoryProofData`.

Eco: Fixed in PR 120.

Cantina Managed: Reviewed that PR 120 fixes the issue as recommended.

3.1.2 Anyone can bypass `faultDisputeGameIsResolved()` storage proofs to drain protocol escrowed funds for Cannon-specific chains

Severity: Critical Risk

Context: Prover.sol#L405-L426

Description: Anyone can bypass storage proofs for `L2_FAULT_DISPUTE_GAME_ROOT_CLAIM_SLOT` and `L2_-FAULT_DISPUTE_GAME_STATUS_SLOT` in `faultDisputeGameIsResolved()` by controlling both the proof and root arguments in `SecureMerkleTrie.verifyInclusionProof()` of `proveStorage()`. This can be abused to drain protocol escrowed funds for Cannon-specific destination chains.

`proveWorldStateCannon()` allows anyone (ideally solvers or their set claimants) to validate world state on a source chain for Cannon-based Optimism L2 destination chains where they have fulfilled intents. This is a required step before they can call `proveIntent()` on a specific intent that they have fulfilled and are attempting to claim its rewards. `proveWorldStateCannon()` implements the following sequence of sub-steps:

1. Validates that the `l1WorldStateRoot` argument is indeed the `existingSettlementBlock-Proof.stateRoot` for the `chainId` argument's settlement chain.
2. Validates that the `l2WorldStateRoot` argument corresponds to a resolved `DEFENDER_WINS` valid fault dispute game that has been generated by the L2's `settlementContract` on L1 rooted at `l1WorldStateRoot` from (1). This is done in 2 parts:
    1. `_faultDisputeGameFromFactory()` performs an account proof for `disputeGameFactoryAddress` and a storage proof for `disputeGameFactoryStorageSlot` on the provided gameID to return `faultDisputeGameProxyAddress` and `rootClaim` (which includes `l2WorldStateRoot`).
    2. `faultDisputeGameIsResolved()` performs an account proof for `faultDisputeGameProxyAd-dress`, a storage proof for `L2_FAULT_DISPUTE_GAME_ROOT_CLAIM_SLOT` on `rootClaim` and another storage proof for `L2_FAULT_DISPUTE_GAME_STATUS_SLOT` on `faultDisputeGameStatusStorage`.
3. Encodes `rlpEncodedBlockData` argument into a `blockProof` to store the proved `l2WorldStateRoot` along with block number and hash (which are derived from `rlpEncodedBlockData`) only if `exist-ingBlockProof.blockNumber < blockProof.blockNumber` so that only successive L2 blocks can be proved.

However, in step (2)(b), the two storage proofs use `faultDisputeGameProofData.faultDisputeGameRootClaimStorageP` for proof and `faultDisputeGameProofData.faultDisputeGameStateRoot` for root. Both the proof and root are derived from the `faultDisputeGameProofData` parameter of `proveWorldStateCannon()`, which is controlled by the caller. This allows anyone to provide a proof that always satisfies their root which has not been verified earlier against anything else that has been proved. This effectively lets anyone bypass these two storage proofs for verifying that a particular L2 block is valid and that its fault dispute game has been resolved.

Impact: High, because this allows bypass of storage proofs for arbitrary L2 blocks that can contain fake transactions with fake fulfillment of any intents with chosen claimants, i.e. allows the use of `proveIn-tent()` to claim rewards for any intents irrespective of their fulfillment status--Protocol escrowed funds for Cannon-specific destination chains can be drained.

Likelihood: High, because this allows anyone to bypass `proveWorldStateCannon()` with an arbitrary `l2WorldStateRoot` which can further be used to bypass the check in `proveIntent()`.

Note: This is independent of "Anyone can arbitrarily set `provenStates[chainId].blockNumber` in `proveWorld-StateCannon()` to prevent proving of all intents" and would be valid even if that were fixed as recommended.

Recommendation: Instead of using `faultDisputeGameProofData.faultDisputeGameStateRoot` (controlled by the caller), consider using `bytes32 faultDisputeGameStorageRoot = bytes32(RLPReader.readBytes(RLPReader.readList(rlpEncodedFaultDisputeGameData)[2]))`.
This will bind the storage proof root used in both storage proofs of (2)(b) to `rlpEncodedFaultDis-puteGameData`, which has been verified in `proveAccount()` for `faultDisputeGameProxyAddress` in (2)(b).

Eco: Fixed in PR 121.

Cantina Managed: Reviewed that PR 121 fixes the issue as recommended.

3.2 High Risk

3.2.1 Invalid RLP encoding causes proof verification failures for values with leading zeros

Severity: High Risk

Context: Prover.sol#L498-L538

Description: The Prover contract incorrectly prepends fixed-length RLP encoding prefixes to values being proven, causing proof verification to fail when values contain leading zeros. This affects both L2 state root verification and reward claiming, potentially leaving funds permanently stuck if a claimant's address contains leading zeros.

In the `proveStorage()` function, the contract uses fixed-length prefixes (e.g., `0xa0`, `0x94`) when encoding values for RLP verification:

```
proveStorage(
    abi.encodePacked(messageMappingSlot),
    bytes.concat(hex"94", bytes20(claimant)), // Fixed 0x94 prefix
    l2StorageProof,
    bytes32(inboxStateRoot)
);
```

However, RLP encoding requires dynamic length prefixes that depend on both the length and content of the value being encoded. When a value contains leading zeros, using a fixed-length prefix causes the proof verification to fail because:

1. The fixed prefix incorrectly indicates a specific length regardless of leading zeros.
2. The actual value's encoding should strip leading zeros and use a prefix based on the resulting length.
3. The mismatch between the encoded value and its merkle proof causes verification to fail.

The issue manifests in three scenarios:

1. L2 state root verification -- If roots contain leading zeros, proofs will fail until a root without leading zeros is available.
2. Reward claiming -- If a claimant's address contains leading zeros, their rewards become permanently unclaimable.
3. Game type encoding:
    - For the default `CANNON` game type (0), the code uses a 24-byte encoding.
    - The length prefix then depends on the timestamp value.
    - When the timestamp exceeds 2^{32} (around February 2106), this encoding will break.

Impact: The impact can be high for affected cases:

- Claimants with addresses containing leading zeros will be unable to claim their rewards permanently.
- Bedrock proofs will fail if the `outputRoot` contains leading zeros.
- Cannon proofs will fail if the `rootClaim` contains leading zeros.
- Cannon proofs will break in 2106 when timestamps exceed 32 bits.

For L2 state root verification, failures could delay proof verification beyond timeouts, disrupting the protocol's operation.

Likelihood: The likelihood of encountering this issue is moderate:

- Claimant addresses have a 1/256 chance (~0.4%) of containing leading zeros.
- Bedrock world state proofs have a 1/256 chance of failure when the `outputRoot` contains leading zeros.
- Cannon proofs have a 1/256 chance of failure when `rootClaim` contains leading zeros.
- The issue is guaranteed to affect all proofs after February 2106 due to the changed length prefix of the encoded game ID.

Proof of Concept:

```
it('unable to verify SLOT3 with fixed length prefix', async () => {
    const slot = 3
    const { key, value, proof, hash } = await getStorageProof(storage, slot)

    const lengthPrefix = '0xa0'
    const valueRlp = lengthPrefix + zeroPadValue(toBeHex(value), 32).slice(2)

    expect(value).to.eq('0x407ef388ae4cde1f592306c95')
    expect(valueRlp).to.eq(
      '0xa000000000000000000000000000000000000000000407ef388ae4cde1f592306c95',
    )

    const valid = await verifyStorageProof(prover, key, valueRlp, proof, hash)
    expect(valid).to.be.false
})
```

Recommendation: Replace fixed-length prefix concatenation with proper RLP encoding using `RLPWriter.writeUint()`:

```
- bytes.concat(hex"94", bytes20(claimant))
+ RLPWriter.writeUint(uint160(claimant))
```

This ensures values are correctly RLP encoded with dynamic length prefixes that handle leading zeros properly. The change should be made everywhere `proveStorage()` is called with manually constructed RLP values.

*Note that **only** `RLPWriter.writeUint()` strips leading zeros from the encoded value. `RLPWriter.writeBytes()` and `RLPWriter.writeAddress()` should not be used.*

Eco: Fixed in PR 139.

Cantina Managed: Reviewed that PR 139 fixes the issue as recommended.

3.2.2 Malicious intent creator can provide arbitrary reward tokens to drain/deny solver rewards after intent fulfillment

Severity: High Risk

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

Description: A malicious intent creator can provide arbitrary reward tokens, while creating the intent in `createIntent()`, which denies solver rewards even after they have fulfilled the intent.

An intent creator is expected to create intents on the source chain via a call to `createIntent()`, which includes reward tokens `_rewardTokens` and amounts `_rewardAmounts` for the intent solver.

However, there are multiple validations missing for this aspect as itemized in the below sub-issues:

1. A malicious intent creator can provide arbitrary reward tokens.
2. Reward tokens are transferred from the intent creator to `IntentSource` using `transferFrom` assuming that they are typical ERC20 tokens that do not require special handling. Return value of `transferFrom` is not checked which will cause an issue with tokens that do not revert but return `false` on unsuccessful transfers (for e.g. ZRX, see the weird-erc20 github repository).
3. Token balances before and after the transfer are not checked for any fee-on-transfer tokens that may be used.
4. The reverting check for `NoRewards` considers the zero-length of `_rewardToken` but discounts potential zero-values of `_rewardAmounts`.

Impact: High, because the above listed missing validations allow a malicious intent creator to provide arbitrary reward tokens/amounts to drain/deny solver rewards after intent fulfillment. Guaranteed reward payment to solvers is a critical fairness aspect of intent fulfillment, which can be circumvented by malicious intent creators.

- Example Scenario: Use of reward tokens that do not revert on unsuccessful transfers allow malicious intent creators to drain the protocol of any such token balances deposited by other intent creators.

1. Alice, an honest intent creator, creates an intent with 100 ZRX reward tokens (ZRX token returns true/false status on transfers without reverting), which are escrowed by `IntentSource`.
2. Bob comes along and creates a malicious intent also for 100 ZRX which he does not have. Because the transfer success status is not checked, his intent is still created without him transferring any ZRX tokens.
3. Bob can then solve his own intent and withdraw ZRX reward tokens that were deposited by Alice.
4. Carol who solves Alice's intent is unable to withdraw Alice's deposited ZRX rewards because they have been drained by Bob in (3). Bob thereby denies Carol the solver rewards.

Likelihood: Medium, because intent creators are game-theoretically motivated to getting their intents filled by providing less/no rewards to solvers. Per protocol assumption, solvers are expected to be sophisticated entities which will somehow validate all such aspects of reward tokens/amounts before attempting to fulfill them. In the above example, even if Carol validates Alice's intent in (1) which appears valid, she may be front-run by Bob in steps (2) & (3) leading to her rewards being denied in (4).

Recommendation:

1. Consider a whitelist of rewards tokens as part of an initial guarded launch.
2. Consider using OpenZeppelin's `SafeERC20` functions to handle transferring of atypical ERC20 reward tokens.
3. Consider adding a defensive balance check before and after the transfer in `createIntent()` to account for any fee-on-transfer tokens.
4. Consider adding a defensive zero-value check for `_rewardAmounts` as part of the reverting `NoRewards` check.

Eco: Fixed in PR 111 and PR 117 for sub-issue and recommendation (2).

Acknowledged sub-issues and recommendations (1), (3) and (4):

1. Didn't want to do a whitelist to keep it as open as possible.
2. There are no fee-on-transfer tokens we're focused on right now, but the solver is charged with ensuring that all token-related risks are sorted out.
3. Responsibility delegated to solvers.

Cantina Managed: Reviewed that PR 111 and PR 117 resolve the transferring of atypical ERC20 reward tokens as recommended. Acknowledged responses for (1), (3) and (4).

3.3 Medium Risk

3.3.1 State proof dependencies enable denial of service attacks

Severity: Medium Risk

Context: Prover.sol#L270-L330

Description: The Prover contract's L2 world state proofs depend on previously proven L1 states. Due to the contract only storing the latest proven state for each chain, a malicious actor can invalidate in-progress L2 proofs by rapidly submitting newer L1 state proofs, effectively denying service to other provers.

The Prover contract uses a hierarchical proof system where:

1. L2 world state proofs depend on proven L1 states via `provenStates[settlementChainId]`.
2. Intent proofs depend on proven L2 states.
3. Only the latest proven state is stored for each chain in `provenStates`.

This creates an opportunity for a malicious actor to disrupt proving by:

1. Waiting for multiple unproven L1 blocks to accumulate.
2. Detecting when someone starts creating an L2 proof targeting a specific L1 block.
3. Quickly submitting a proof for a newer L1 block before the L2 proof completes.

4. Causing the L2 proof to fail when submitted since its target L1 state is no longer the latest.

Impact: If executed correctly this vulnerability could allow malicious actors to temporarily block L2 state proofs and consequently intent proofs. A legitimate prover could be unaware of the reasons for their proofs not succeeding. In a worst case scenario, the intent rewards could be returned to the intent creator once the expiry time is reached.

Likelihood: The likelihood of exploitation is low and depends on several factors:

- Speed of L2 proof generation vs L1 block time.
- Number of accumulated unproven L1 blocks available.
- Predictability of when L2 proofs will be submitted.

Currently, L2 proof generation (1-2 seconds) is faster than L1 block time (~12 seconds), making exploitation harder but still possible by accumulating multiple unproven blocks.

Recommendation: Consider implementing the following mitigations:

1. Store a history of proven states instead of only the latest:

```diff
- mapping(uint256 => BlockProof) public provenStates;
+ mapping(uint256 chainId => mapping(uint256 blockNumber => BlockProof)) public provenStates;
+ mapping(uint256 chainId=> uint256 blockNumber) public latestProvenStateBlockNumber;
```

2. Allow L2 proofs to target any previously proven L1 state.
3. Ensure the claimant for an already proven intent cannot be overridden.

These changes would make the system more robust against proof invalidation attacks.

Eco: Fixed in PR 113.

Cantina Managed: Ideally the delay would apply to the timestamp/block number of when the last proof was submitted instead of adding a delay between subsequent blocks to be proven. Would disallowing proving "stale blocks", i.e. blocks that are older than X (e.g. 20) blocks be a possibility in addition to this (and maybe lowering the delay requirement). This would also further limit the possibility of building up a queue of blocks that can be proven.

Eco:

> Ideally the delay would apply to the timestamp/block number of when the last proof was submitted instead of adding a delay between subsequent blocks to be proven.

Yes, we could add blockTimeStamp to the worldState and may consider doing this moving forward. However, I believe by using `SETTLEMENT_BLOCKS_DELAY`, we can achieve the same result, albeit with a slight variance if block times differ on the settlement chain. That variance, I believe, does not open up attack vectors.

> Would disallowing proving "stale blocks", i.e. blocks that are older than X (e.g. 20) blocks be a possibility in addition to this (and maybe lowering the delay requirement). This would also further limit the possibility of building up a queue of blocks that can be proven.

Proving the latest block effectively prevents any earlier "stale blocks" from being proven. So whilst we could also add an additional check to prevent "stale blocks" it may be unnecessary.

Cantina Managed: Using Block delays would also be an option, however as you mentioned these would have to be adjusted depending on the chain and therefore add a bit more mental complexity to the picture.

The outlined attack scenario requires an attacker to accumulate, say 20 blocks from the settlement chain which are ready to be proven. Instead of simply proving the latest block, the attacker proves the 20 blocks in sequence (every time an honest actor wants to submit an intent fulfillment proof) one by one thereby enabling the described DoS scenario. If we require a minimum delay of 5 blocks, the attacker will now have 4 DoS opportunities (every 5 blocks) if 20 blocks are unproven from the settlement chain. While this reduces the likelihood, the attacker can now wait for, e.g. 100 unproven blocks (not sure if realistic) to reach 20 DoS opportunities again. If there was, however, a check to prevent proving "stale" blocks that lie too far in the past, then the attacker might not be able to accumulate 100 unproven blocks.

I realize that this would require checking the settlement chain's timestamp, since block numbers from the source chain cannot be compared to the settlement chain.

Eco: Assuming:

1. The black hat is doing the DoS attack.
2. The white hat is the client submitting the valid last block (and also potentially a valid L2 world state proof).

If the white hat submits the latest valid block it would then cease the DOS attack as the black hat would no longer be able to submit outdated blocks, how would the black hat stop the white hat from submitting the latest valid block and thus invalidating their DOS attack?

To be clear there must always be a white hat submitting the valid L1 state which is required for the L2 state. Usually (but not necessarily) it is the some user.

Cantina Managed: Confirming that this mitigation is valid, however it relies on off-chain procedures and assumptions which are not being enforced on-chain.

3.4 Low Risk

3.4.1 Chain ID should not be stored as immutable constant

Severity: Low Risk

Context: IntentSource.sol#L39-L43

Description: The `IntentSource` contract stores the chain ID as an immutable constant during contract deployment. This design choice could lead to issues in case of a chain fork, as the stored chain ID would no longer match the actual chain ID of the forked chain. The mismatch would affect the generation of nonces and intent hashes in the `createIntent()` function, potentially breaking cross-chain functionality.

```
bytes32 _nonce = keccak256(abi.encode(counter, CHAIN_ID));
bytes32 intermediateHash =
    keccak256(abi.encode(CHAIN_ID, _destinationChainID, _targets, _data, _expiryTime, _nonce));
```

In the event of a chain fork, all intents created after the fork would use the old chain ID, leading to invalid intent hashes and potential cross-chain message delivery issues.

Recommendation: Consider using `block.chainid` directly in the hash calculations instead of storing it as an immutable variable:

```
  contract IntentSource is IIntentSource {
-     uint256 public immutable CHAIN_ID;

      constructor(uint256 _minimumDuration, uint256 _counterStart) {
-         CHAIN_ID = block.chainid;
          MINIMUM_DURATION = _minimumDuration;
          counter = _counterStart;
      }

      function createIntent(...) external payable {
          // ...
-         bytes32 _nonce = keccak256(abi.encode(counter, CHAIN_ID));
+         bytes32 _nonce = keccak256(abi.encode(counter, block.chainid));
          bytes32 intermediateHash =
-             keccak256(abi.encode(CHAIN_ID, _destinationChainID, _targets, _data, _expiryTime, _nonce));
+             keccak256(abi.encode(block.chainid, _destinationChainID, _targets, _data, _expiryTime, _nonce));
          // ...
      }
  }
```

Eco: Fixed in PR 111.

Cantina Managed: Reviewed that PR 111 fixes the issue as recommended.

3.4.2 Missing payable modifier for fulfillment functions requiring native token transfers

Severity: Low Risk

Context: Inbox.sol#L185-L194

Description: The `fulfillStorage()` and `fulfillHyperBatched()` functions in the Inbox contract lack the `payable` modifier, despite potentially needing to execute calls that transfer native tokens through the contract's `transferNative()` function. This oversight could prevent the fulfillment of intents that include native token transfers in their execution path.

The issue arises because:

1. The `transferNative()` function is designed to allow native token transfers during intent execution.
2. Intents executed via `_fulfill()` can include calls to `transferNative()` requiring value to be located in the contract.
3. Without the `payable` modifier, these functions cannot receive the native tokens needed for these transfers.

Recommendation: Add the `payable` modifier to both functions to allow them to receive native tokens needed for transfers:

```diff
  function fulfillStorage(
      uint256 _sourceChainID,
      address[] calldata _targets,
      bytes[] calldata _data,
      uint256 _expiryTime,
      bytes32 _nonce,
      address _claimant,
      bytes32 _expectedHash
- ) external returns (bytes[] memory) {
+ ) external payable returns (bytes[] memory) {

  function fulfillHyperBatched(
      uint256 _sourceChainID,
      address[] calldata _targets,
      bytes[] calldata _data,
      uint256 _expiryTime,
      bytes32 _nonce,
      address _claimant,
      bytes32 _expectedHash,
      address _prover
- ) external returns (bytes[] memory) {
+ ) external payable returns (bytes[] memory) {
```

Eco: Fixed in PR 111.

Cantina Managed: Reviewed that PR 111 fixes the issue as recommended.

3.4.3 L1 `blockhashoracle` set incorrectly for remote chain configurations

Severity: Low Risk

Context: Prover.sol#L114-L119, Prover.sol#L127-L130

Description: In the Prover contract, the `_setChainConfiguration()` function updates the `l1BlockhashOracle` address for every chain configuration processed in the constructor loop. This is incorrect as the `l1BlockhashOracle` should only be set for the local chain (current L2 chain where the contract is deployed), since it needs to interact with a contract that exists on the same chain.

The current implementation causes the `l1BlockhashOracle` to be set to the last chain's oracle address in the configuration array, which may be an invalid address if that chain is not the local chain. This could cause `proveSettlementLayerState()` to revert, preventing proving and withdrawals.

Recommendation: Modify `_setChainConfiguration()` to only set the `l1BlockhashOracle` for the local chain:

```diff
  function _setChainConfiguration(uint256 chainId, ChainConfiguration memory chainConfiguration) internal {
      chainConfigurations[chainId] = chainConfiguration;
-     l1BlockhashOracle = IL1Block(chainConfiguration.blockhashOracle);
+     if (chainId == block.chainid) {
+         l1BlockhashOracle = IL1Block(chainConfiguration.blockhashOracle);
+     }
  }
```

Additionally, consider adding a check in the constructor to ensure that the local chain's configuration is included to avoid misconfiguration:

```diff
  constructor(ChainConfigurationConstructor[] memory _chainConfigurations) {
+     bool localChainConfigured;
      for (uint256 i = 0; i < _chainConfigurations.length; ++i) {
          _setChainConfiguration(_chainConfigurations[i].chainId, _chainConfigurations[i].chainConfiguration);
+         if (_chainConfigurations[i].chainId == block.chainid) {
+             localChainConfigured = true;
+         }
      }
+     if (!localChainConfigured) {
+         revert LocalChainNotConfigured();
+     }
  }
```

Eco: Updated logic to set only for source chain on commit 236ca3ba. Will revisit the checking to ensure there is chain configuration for the current chain:

- Not all chains will have an L1 Block Oracle (only optimism chains).
- Need to have a default value for non-optimistic chains and update `proveSettlementLayerState` to cater for non op chains.

Currently Prover contracts are deployed by ECO and the chain configuration validation is a manual process.

Cantina Managed: Looks good. Just one last remark: So all the `chainConfiguration.blockhashOracle` settings for other chains will be discarded/ignored on the current chain, but still passed in as a parameter?

3.4.4 Malicious intent creator can provide arbitrary prover to deny solver rewards after intent fulfillment

Severity: Low Risk

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

Description: A malicious intent creator can provide an arbitrary/malicious prover contract, while creating the intent in `createIntent()`, that prevents reward withdrawals for solvers after intent fulfillment.

An intent creator is expected to create intents on the source chain via a call to `createIntent()`, which includes the prover address against which the intent's status is checked during `withdrawRewards()`. This `_prover` address is included in the intent and emitted in `IntentCreated`.

However, there is no validation in `createIntent()` to check that the specified `_prover` contract is indeed one of the protocol-deployed valid prover contracts on the source chain. Solvers fulfilling intents created with arbitrary prover contracts may be unable to have their intended claimants withdraw rewards in `withdrawRewards()` if malicious prover contracts, for example, do not perform accurate storage proofs, never set `provenIntents[intentHash]` to the provided claimant or set the claimant to an address other than what the solver provided/intended.

Impact: High, because a malicious intent creator can provide an arbitrary prover to deny solver rewards after intent fulfillment. Guaranteed reward payment to solvers is a critical fairness aspect of intent fulfillment, which can be circumvented by malicious intent creators.

Likelihood: Very Low, because solvers can be reasonably expected to check if the intent provided prover address is one of the valid protocol-deployed contracts. Per protocol assumption, solvers are expected to be sophisticated entities which will validate prover contracts via, for example, a protocol-provided offchain whitelist.

Recommendation: Consider enforcing a whitelist of protocol-deployed provers in `createIntent()` as part of an initial guarded launch. This could be similar to the `solverWhitelist` approach enforced for solvers in Inbox.

Eco: We don't want to do a whitelist for this, but we acknowledge this risk and are expecting solvers to take it on in the early stages.

Cantina Managed: Acknowledged.

3.4.5 Malicious intent creator can provide `_expiryTime` that is sufficient only for solver intent execution but not proving

Severity: Low Risk

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

Description: A malicious intent creator can provide an `_expiryTime`, while creating the intent in `createIntent()`, that is only sufficient for solver intent execution but not proving fulfillment for reward withdrawal. This allows intent creator to withdraw their provided rewards themselves, after the intent expires, to effectively get free intent execution at the expense of the solver.

An intent creator is expected to create intents on the source chain via a call to `createIntent()`, which includes the `_expiryTime`. As commented:

> If a proof ON THE SOURCE CHAIN is not completed by the expiry time, the reward funds will not be redeemable by the solver, REGARDLESS OF WHETHER THE INSTRUCTIONS WERE EXECUTED. The onus of that time management (i.e. how long it takes for data to post to L1, etc.) is on the intent solver.

While `_expiryTime` is checked to be at least of the configured `MINIMUM_DURATION` from intent creation, it may be hard to factor-in all possible potential sources of delays possible during proving fulfillment such as L2 data posting to L1, L2 sequencers, L1 block hash oracles, L2 fraud proof challenges and Hyperlane messaging.

Intent creators are allowed to withdraw their posted intent rewards themselves in `withdrawRewards()` after intent expiration even if their intents were executed by solvers but were yet to be proved due to potential delays.

Impact: High, because a malicious intent creator can provide a low `_expiryTime` to deny solvers their rewards even after intent fulfillment. Guaranteed reward payment to solvers is a critical fairness aspect of intent fulfillment, which can be circumvented by malicious intent creators.

Likelihood: Very Low, because, solvers are reasonably expected to check if the intent provided `_expiryTime` is sufficient in worst-case scenarios of potential sources of delays for intent proving. Per protocol assumption, solvers are expected to be sophisticated entities which will validate sufficient intent expiry times along with other intent creation parameters.

Recommendation: Consider enforcing a `MINIMUM_DURATION` that accounts for worst-case scenarios of potential sources of delays.

Eco: The checks in the contract will never be sufficient to inform a potential solver on whether or not they should solve a given intent, so any solver would be expected to do their own validation of expiry times. Given that this check doesn't change behavior, we have elected to remove MINIIMUM_DURATION altogether to prevent confusion -- the expiry check is now entirely in the solvers' hands, in line with our expectations of that persona

Cantina Managed: Acknowledged.

3.4.6 Deleted L2 outputs remain provable after deletion

Severity: Low Risk

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

Description: The `L2OutputOracle`'s `deleteL2Outputs()` function only truncates the outputs array length without clearing individual elements. As a result, deleted L2 outputs remain in contract storage and can still be proven valid through `proveWorldStateBedrock()`, potentially allowing the use of invalidated state roots.

When L2 outputs are deleted in the `L2OutputOracle` contract, the implementation only updates the array length in storage:

```
assembly {
    sstore(l2Outputs.slot, _l2OutputIndex)
}
```

This has the effect that:

1. The deleted L2 output data remains in contract storage.
2. The storage proofs for these "deleted" outputs remain valid since the data hasn't been cleared.

The Prover contract's `proveWorldStateBedrock()` function validates L2 outputs through storage proofs but doesn't verify if the output index is within the current valid range. This means invalidated outputs can still be proven valid since:

1. Their data still exists in storage.
2. No array bounds checking is performed.
3. The storage proof will succeed.

Impact: The impact is high as it allows the use of L2 state roots that have been explicitly invalidated through deletion. This could create inconsistencies between L2 state verification on different chains and allow proving of intents that were fulfilled during invalidated blocks. If an invalid L2 state were to become verified, an attacker could potentially build upon that invalid state.

Likelihood: The likelihood of this vulnerability being exploitable is low because:

1. Output deletion is considered an emergency operation and is restricted to the challenger address.
2. The deletion operation itself cannot be triggered maliciously.
3. An invalidated L2 state must be proven before it is overwritten.
4. An intent must be fulfilled in an invalidated block to cause issues.

Recommendation: Consider adding array length verification by modifying the `proveWorldStateBedrock` function to validate the L2 output index:

```
  function proveWorldStateBedrock(
      uint256 chainId,
      bytes calldata rlpEncodedBlockData,
      bytes32 l2WorldStateRoot,
      bytes32 l2MessagePasserStateRoot,
      uint256 l2OutputIndex,
      bytes[] calldata l1StorageProof,
+     bytes[] calldata lengthProof,
+     uint256 l2OutputsArrayLength,
      bytes calldata rlpEncodedOutputOracleData,
      bytes[] calldata l1AccountProof,
      bytes32 l1WorldStateRoot
  ) public virtual {
      ChainConfiguration memory chainConfiguration = chainConfigurations[chainId];
      BlockProof memory existingSettlementBlockProof = provenStates[chainConfiguration.settlementChainId];
      require(
          existingSettlementBlockProof.stateRoot == l1WorldStateRoot, "settlement chain state root not yet
↪  proved"
      );

+     // Prove and verify array length
+     proveStorage(
+         abi.encodePacked(L2_OUTPUT_SLOT_NUMBER),
+           RLPWriter.writeUint(l2OutputsArrayLength),
+           lengthProof,
+           bytes32(outputOracleStateRoot)
+       );
+       require(l2OutputIndex < l2OutputsArrayLength, "l2OutputIndex exceeds length");
+
        bytes32 outputRootStorageSlot =
            bytes32(abi.encode((uint256(keccak256(abi.encode(L2_OUTPUT_SLOT_NUMBER))) + l2OutputIndex * 2)));

        bytes memory outputOracleStateRoot =
↪    RLPReader.readBytes(RLPReader.readList(rlpEncodedOutputOracleData)[2]);

        require(outputOracleStateRoot.length <= 32, "contract state root incorrectly encoded"); // ensure
↪    lossless casting to bytes32
        proveStorage(
            abi.encodePacked(outputRootStorageSlot),
        proveStorage(
            abi.encodePacked(outputRootStorageSlot),
            bytes.concat(bytes1(uint8(0xa0)), abi.encodePacked(outputRoot)),
            l1StorageProof,
            bytes32(outputOracleStateRoot)
        );

        // ... rest of the function
```

However, given the low likelihood and additional complexity these changes would introduce, it may be reasonable to accept this risk and clearly document and monitor the situation.

Eco: Addressed in PR 122 by adding in a `finalityDelayPeriod`.

Cantina Managed: The fix now requires a delay in the block timestamp of the Bedrock proof. This does not guarantee that the new block has passed the finality delay as seen from the settlement chain. Ideally, the delay would measure the time delay of the new Bedrock proof and the last settlement proof. Currently `BlockProof` only stores `blockNumber`. `blockNumber` and `blockTimestamp` could easily be packed together in the struct at no extra storage costs. Or block number delays could be specified instead.

Eco: We are aligning with the Destination chain's FINALIZATION_PERIOD_SECONDS, which is what the chain has in place to ensure that they are past any potential re-org. This only applies to chains using Bedrock Proving Mechanisms (not Cannon).

An example of this is Mantle which has a finalization period of 1 week (604800 seconds) (see contract 0x31d543e7BE1dA6eFDc2206Ef7822879045B9f481.

So by validating that any proof submitted has the latest block being settled on the destination chain of a week, we prevent any potential re-orgs. In this scenario, I do not see how checking last proof time addresses this.

For example batches are posted every hour, however they are not finalized till a week later. I don't see how checking how long since the last batch was posted mitgates the attack vector of a reorg. This does require that at time of deployment the `finalityDelaySeconds` is aligned to that of the destination chains value.

Can you clarify, are you highlighting that this alignment introduces risks and you'd like to mitigate those by doing a storage proof of the destination chains FINALIZATION_PERIOD_SECONDS as part of the constructor?

Cantina Managed: We'd like to point out a few assumptions for this mitigation to be valid:

1. After the finality delay, any previously deleted L2Block will always be overwritten by new L2Blocks, and therefore will not be available in the L1 state root anymore. So it can't be proven anymore.
2. The L1 state root is not stale.

Under these assumptions, I believe the fix should be ok.

For counter scenarios, imagine the following edge cases:

1. An invalid L2Block root is stored in the L2 block hash oracle. The block hash oracle stops committing new blocks. After the finalization period, the L2 block (although deleted by an admin) is still provable.
2. The last L1 root proof is past the finality delay. That means that deleted roots might still be provable in the last L1 state root (even though on-chain on the L1 it might be removed from storage).

My original response was a bit confusing and not entirely accurate. I believe that assuming 1) will hold is fine. However, I believe that for 2) checking that the last settlement proof is not stale should be required. And this would require comparing timestamps of the last settlement proof to not be older than the finality delay (and perhaps a small additional margin dependent on the frequency of the output oracle).

We understand that in the normal L1 Ethereum case proofs might be quite frequent, but I'm also thinking of the L2 $\rightarrow$ L3 scenario.

Eco: In response to the counter scenarios above...

1. We are aligning the `FINALIZATION_PERIOD_SECONDS` with that of the chain (i.e. 604800 seconds or a week) in Mantle's case. If the chain has not settled any batches for a week (or whatever their stated finalization period), then this is a larger problem that the chain is not functioning correctly.
2. Once again in this scenario. If the chain has not settled any batches for a week (or whatever their stated finalization period),then this is a larger problem that the chain is not functioning correctly.

Cantina Managed: We'd acknowledge scenario 1) if these settings line up and agree that the chain would have a larger problem then. Regarding 2), the scenario describes when the source chain, e.g. Base, wants to use a Bedrock proof for e.g. Mantle, yet the settlement chain's latest block proof is stale. This does not require the Mantle network to halt block production, it merely requires the last L1 proof to be not up to date.

Eco: For scenario 2 in order for this attack vector to happen.

1. The destination chain (e.g. Mantle) has to have undergone a re-org.
2. The L1 World State (settlement Proof) has to have not been updated for greater than the `FINALIZATION_PERIOD_SECONDS` i.e. a week.

If this was the case then a proof could be done for the destination chain for a block that has been re-orged.

i.e. we are relying on, in the case of a re-org, that the source chains L1 World State (settlement proof) has been updated within the last week (`FINALIZATION_PERIOD_SECONDS` i.e. a week.).

We do agree that this is a possible attack vector. However, I think is acceptable, as we are currently proving L1 world state typically when doing any destination chain L2 proof (Bedrock or Cannon) which translates to an hour.

Also in the unusual case of a re-org. The simplest counter is to just prove the L1 World State (settlement proof) on the source chains that have this chain as a target chain. Thus ensuring the state is not stale.

3.4.7 Excess fee is not refunded back to solvers using Hyperlane

Severity: Low Risk

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

Description: Solvers are expected to pay fees with native tokens when they utilize Hyperlane mailbox for instant proving of their fulfilled intents. However, any excess fee paid by them is lost to the protocol and not refunded back to them.

Solvers in the protocol have an option to use Hyperlane mailbox architecture for instant proving of their fulfilled intents. This is an alternative to proving their fulfillment using storage proofs which is trustless but slower. Hyperlane introduces another trust assumption and also charges a fee. So, while using Hyperlane intent proving via `mailbox.dispatch` in `fulfillHyperInstantWithRelayer()` or `sendBatchWithRelayer()`, provers are expected to pay fees via native tokens. The exact fee is determined during the call using Hyperlane `mailbox.quoteDispatch()`.

However, any excess `msg.value` sent by a prover towards Hyperlane fee is not refunded back but instead is left behind in the protocol to be later extracted by protocol Inbox owner via `drain()`. A prover sending any excess fee therefore loses it to the protocol.

Impact: Low, because it is expected that excess fees will not be significant amounts.

Likelihood: Low, because it is expected that solvers will determine appropriate fees for such calls and not send much in excess.

Recommendation: Consider refunding `msg.value - fee` using `call` instead of letting such dust amount accumulate in the protocol for later draining.

Eco: Fixed in PR 118 and PR 131.

Cantina Managed: Reviewed that PR 118 and PR 131 resolve the issue as recommended. Adherence to CEI convention could be considered.

3.4.8 Anyone can drain native tokens from Inbox using `transferNative()`

Severity: Low Risk

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

Description: Anyone can drain leftover native tokens from Inbox using `transferNative()` circumventing the privileged `drain()` functionality.

Inbox implements `transferNative()` to allow intent creators to transfer native tokens on destination chain. One possible sequence of actions is:

1. Intent creator Alice includes a `transferNative()` call as one of the intent targets with its intended `_to` recipient on the destination chain. The intended `_amount` is included as part of the intent rewards on the source chain.
2. Intent solver Bob decides to fulfill Alice's intent and first transfers `_amount` native tokens to Inbox on destination chain.
3. Bundled with (2) for atomicity, Bob fulfills Alice's intent which includes the `transferNative(_to, _amount)` call.

Any excess native tokens sent by Bob will remain as dust within Inbox. Bob may also decide to use the Hyperlane prover for faster execution and therefore includes native tokens for Hyperlane fee. Any excess fee will also remain within Inbox. For withdrawing such dust and excess fee native tokens, Inbox implements a privileged `onlyOwner` function `drain()`. However, anyone on the destination chain can execute `fulfillStorage()` with arbitrary values of `_fulfill()` parameters to satisfy implemented checks to execute `transferNative(_to, _amount)` as one of the target calls and thereby transfer any leftover native tokens from Inbox.

Impact: Low, because Inbox is not expected to have significant escrowed amounts of native tokens. However, the implementation of `drain()` implies that protocol expects to have some dust and excess fee native tokens.

Likelihood: Medium, because anyone can circumvent the privileged `drain()` functionality to drain any leftover native tokens.

Recommendation: 1. Reconsider the design of `transferNative()` and `drain()`. 2. Consider refunding any excess native tokens to the Solver immediately to leave no excess amounts in Inbox.

Eco: We elected to remove drain() altogether and also refund excess native token in PR 118.

Cantina Managed: Reviewed that PR 118 fixes the issue as recommended by removing `drain()` and refunding excess fee immediately without leaving any dust.

3.5 Gas Optimization

3.5.1 Gas optimizations

Severity: Gas Optimization

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

Description/Recommendation: The following optimizations will reduce gas costs by eliminating redundant operations, minimizing storage reads, and optimizing memory usage.

1. Redundant Operations in `assembleGameStatusStorage()`: The function contains multiple unnecessary `abi.encodePacked` and `bytes.concat` calls and conditional logic that can be simplified. The RLPWriter will automatically handle leading zeros, making the conditional branch unnecessary.

```
  function assembleGameStatusStorage(
      uint64 createdAt,
      uint64 resolvedAt,
      uint8 gameStatus,
      bool initialized,
      bool l2BlockNumberChallenged
  ) public pure returns (bytes memory) {
-     if (l2BlockNumberChallenged) {
-         return bytes.concat(
-             RLPWriter.writeBytes(
-                 abi.encodePacked(
-                     abi.encodePacked(l2BlockNumberChallenged),
-                     abi.encodePacked(initialized),
-                     abi.encodePacked(gameStatus),
-                     abi.encodePacked(resolvedAt),
-                     abi.encodePacked(createdAt)
-                 )
-             )
-         );
-     } else {
-         return bytes.concat(
-             RLPWriter.writeBytes(
-                 abi.encodePacked(
-                     abi.encodePacked(initialized),
-                     abi.encodePacked(gameStatus),
-                     abi.encodePacked(resolvedAt),
-                     abi.encodePacked(createdAt)
-                 )
-             )
-         );
-     }
+     return RLPWriter.writeBytes(
+         abi.encodePacked(
+             l2BlockNumberChallenged,
+             initialized,
+             gameStatus,
+             resolvedAt,
+             createdAt
+         )
+     );
  }
```

2. Redundant keccak256 Operations in Proving Functions: Both `proveSettlementLayerState()` and `proveWorldStateBedrock()` calculate the same hash twice. Caching the result saves gas:

```
  function proveSettlementLayerState(bytes calldata rlpEncodedBlockData) public {
+     bytes32 blockHash = keccak256(rlpEncodedBlockData);
-     require(keccak256(rlpEncodedBlockData) == l1BlockhashOracle.hash(), "hash does not match block data");
+     require(blockHash == l1BlockhashOracle.hash(), "hash does not match block data");

      BlockProof memory blockProof = BlockProof({
          blockNumber: _bytesToUint(RLPReader.readBytes(RLPReader.readList(rlpEncodedBlockData)[8])),
-         blockHash: keccak256(rlpEncodedBlockData),
+         blockHash: blockHash,
          stateRoot: bytes32(RLPReader.readBytes(RLPReader.readList(rlpEncodedBlockData)[3]))
      });
```

3. Unnecessary Memory Copy in `sendBatchWithRelayer()`: The function unnecessarily copies calldata to memory when the calldata array can be used directly:

```
  function sendBatchWithRelayer(uint256 _sourceChainID, address _prover, bytes32[] calldata _intentHashes,
↪  bytes memory _metadata, address _postDispatchHook) public payable {
      // ...
-     bytes32[] memory hashes = new bytes32[](size);
      address[] memory claimants = new address[](size);
      for (uint256 i = 0; i < size; i++) {
-         hashes[i] = _intentHashes[i];
          claimants[i] = fulfilled[_intentHashes[i]];
      }
-     bytes memory messageBody = abi.encode(hashes, claimants);
+     bytes memory messageBody = abi.encode(_intentHashes, claimants);
```

4. Redundant Storage Read in `withdrawRewards` and optional call in `batchWithdraw()`: The function reads `intent.rewardNative` twice when it could be cached:

```
  function batchWithdraw(bytes32[] calldata intentHashes) external {
      // ...
      for (uint256 i = 0; i < intentHashes.length; i++) {
          Intent storage intent = intents[intentHashes[i]];
+         uint256 nativeReward = intent.rewardNative;
-         if (intent.rewardNative > 0) {
-             nativeRewards += intent.rewardNative;
+         if (nativeReward > 0) {
+             nativeRewards += nativeReward;
          }
```

```
    function withdrawRewards(bytes32 _hash) external {
        Intent storage intent = intents[_hash];
        address claimant = SimpleProver(intent.prover).provenIntents(_hash);
        address withdrawTo;
        if (!intent.hasBeenWithdrawn) {
            // ...
            for (uint256 i = 0; i < len; i++) {
                safeERC20Transfer(intent.rewardTokens[i], withdrawTo, intent.rewardAmounts[i]);
            }
+           uint256 nativeReward = intent.rewardNative;
+           if (nativeReward > 0) {
+               payable(withdrawTo).transfer(intent.rewardNative);
+           }
-           payable(withdrawTo).transfer(intent.rewardNative);
            emit Withdrawal(_hash, withdrawTo);
        } else {
            revert NothingToWithdraw(_hash);
        }
    }
```

5. Unnecessary Memory Assignments in `getIntent()`: The function redundantly reassigns values that are already loaded into memory:

```
  function getIntent(bytes32 identifier) public view returns (Intent memory) {
-     Intent memory intent = intents[identifier];
-     intent.targets = intents[identifier].targets;
-     intent.data = intents[identifier].data;
-     intent.rewardTokens = intents[identifier].rewardTokens;
-     intent.rewardAmounts = intents[identifier].rewardAmounts;
-     return intent;
+     return intents[identifier];
  }
```

6. Unused import: The `import "./ISemver.sol";` import in the `SimpleProver` contract is unused, remove it.

Eco: Partially fixed in commits e89ce715, 27fb274a, 5765d337 and 56efdaf6.

Cantina Managed: Reviewed that the changes in the commits implement part of the gas optimization recommendations.

3.6 Informational

3.6.1 Missing validation for prover in `createIntent()`

Severity: Informational

Context: IntentSource.sol#L59-L113

Description: The `createIntent()` function accepts a `_prover` address parameter without performing any validation to ensure it is a valid contract address that can prove intent fulfillment. This could lead to intents being created with invalid or non-existent prover addresses, potentially causing these intents to become unverifiable.

Furthermore, the prover's proof type is not included in the intent hash calculation, missing an opportunity to provide additional information about the intended proving mechanism

The `IProver` interface already defines proof types via the `ProofType` enum and includes a `getProofType()` function. Calling this function would serve both as validation of the prover contract's existence and interface compliance, while also providing the proof type information that could be included in the intent hash.

Recommendation: Consider adding prover validation through the `getProofType()` call and including the returned type in the intent hash:

- `IntentSource`:

```
  function createIntent(
      uint256 _destinationChainID,
      address _inbox,
      address[] calldata _targets,
      bytes[] calldata _data,
      address[] calldata _rewardTokens,
      uint256[] calldata _rewardAmounts,
      uint256 _expiryTime,
      address _prover
  ) external payable {
+     // Fetch proof type - this will revert if prover is invalid
+     // or doesn't implement the interface correctly
+     IProver.ProofType proofType = IProver(_prover).getProofType();

      bytes32 intermediateHash =
          keccak256(abi.encode(
              CHAIN_ID,
              _destinationChainID,
              _targets,
              _data,
              _expiryTime,
              _nonce,
              _prover,
+             proofType
          ));
      bytes32 intentHash = keccak256(abi.encode(_inbox, intermediateHash));

      // ... rest of function ...
  }
```

- `Inbox`:

```
    function fulfillStorage(
        uint256 _sourceChainID,
        address[] calldata _targets,
        bytes[] calldata _data,
        uint256 _expiryTime,
        bytes32 _nonce,
        address _claimant,
        bytes32 _expectedHash,
+       IProver.ProofType _proofType
    ) external payable returns (bytes[] memory) {
+       if (_proofType != IProver.ProofType.Storage) {
+           revert InvalidProverType();
+       }
        // ... rest of function with _proofType included in hash verification ...
    }

    function fulfillHyperInstant(
        // ... existing parameters ...
+       IProver.ProofType _proofType
    ) external payable returns (bytes[] memory) {
+       if (_proofType != IProver.ProofType.Hyperlane) {
+           revert InvalidProverType();
+       }
        // ... rest of function with _proofType included in hash verification ...
    }
```

This change:

1. Validates the prover type on the source chain during intent creation.
2. Includes the proof type in the intent hash.
3. Validates the proof type matches the fulfillment method on the destination chain.

Eco: Acknowledged. We don't think this change would result in any different behavior from the solver. It is not enough to know that the prover exists, or implements the correct interface -- the solver must be sure that the prover works as expected, and they can be assured of this either by inspection or by ensuring that the prover's address matches one that we published. Given that inspecting the prover yields the proof type and that our published address list will arrange provers by proof type, we do not feel that this change is necessary.

Cantina Managed: Acknowledged.

3.6.2 Protocol design improvements

Severity: Informational

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

Description/Recommendation:

1. Return Intent Hash from `createIntent()`: Currently, `createIntent()` does not return the intent hash, requiring integrators to calculate it separately or track it through events. Returning the hash would improve the developer experience and make integrations more straightforward.

```
  function createIntent(
      uint256 _destinationChainID,
      address _inbox,
      address[] calldata _targets,
      bytes[] calldata _data,
      address[] calldata _rewardTokens,
      uint256[] calldata _rewardAmounts,
      uint256 _expiryTime,
      address _prover
-   ) external payable {
+   ) external payable returns (bytes32 intentHash) {
      // ... existing validation and proof type checks ...

      bytes32 intermediateHash = keccak256(abi.encode(CHAIN_ID, _destinationChainID, _targets, _data,
↪   _expiryTime, _nonce));
-     bytes32 intentHash = keccak256(abi.encode(_inbox, intermediateHash));
+     intentHash = keccak256(abi.encode(_inbox, intermediateHash));

      // ... rest of the function ...
+     return intentHash;
  }
```

2. Additional Intent Hash Parameters: The current intent hash calculation could be expanded to include additional parameters for better cross-chain validation and processing. This would allow the destination chain to verify and utilize more information about the intent.

```
  function createIntent(
      uint256 _destinationChainID,
      address _inbox,
      address[] calldata _targets,
      bytes[] calldata _data,
      address[] calldata _rewardTokens,
      uint256[] calldata _rewardAmounts,
      uint256 _expiryTime,
      address _prover
  ) external payable returns (bytes32 intentHash) {
      bytes32 intermediateHash = keccak256(abi.encode(
          CHAIN_ID,
          _destinationChainID,
          _targets,
          _data,
          _expiryTime,
          _nonce,
+         msg.sender,           // Intent creator
+         _prover,             // Prover address
+         _rewardTokens,       // Reward token addresses
+         _rewardAmounts       // Reward amounts
      ));
      intentHash = keccak256(abi.encode(_inbox, intermediateHash));

      intents[intentHash] = Intent({
          creator: msg.sender,
          destinationChainID: _destinationChainID,
          targets: _targets,
          data: _data,
          rewardTokens: _rewardTokens,
          rewardAmounts: _rewardAmounts,
          expiryTime: _expiryTime,
          hasBeenWithdrawn: false,
          nonce: _nonce,
          prover: _prover,
          rewardNative: msg.value
      });

      // ... rest of function ...
      return intentHash;
  }
```

These improvements would enable better validation capabilities and more flexible processing options on the destination chain when fulfilling intents.

3. Adhere to Checks-Effects-Interactions Pattern: The protocol should consistently follow the CEI pattern to prevent potential reentrancy issues. In `withdrawRewards()` and `_fulfill()`, state changes

should (and events could, but not must) occur before external calls:

```
  function withdrawRewards(bytes32 _hash) external {
      Intent storage intent = intents[_hash];
      address claimant = SimpleProver(intent.prover).provenIntents(_hash);
      address withdrawTo;
      if (!intent.hasBeenWithdrawn) {
          if (claimant != address(0)) {
              withdrawTo = claimant;
          } else {
              if (block.timestamp >= intent.expiryTime) {
                  withdrawTo = intent.creator;
              } else {
                  revert UnauthorizedWithdrawal(_hash);
              }
          }
          intent.hasBeenWithdrawn = true;
+         emit Withdrawal(_hash, withdrawTo);

          uint256 len = intent.rewardTokens.length;
          for (uint256 i = 0; i < len; i++) {
              safeERC20Transfer(intent.rewardTokens[i], withdrawTo, intent.rewardAmounts[i]);
          }
-         payable(withdrawTo).transfer(intent.rewardNative);
+         (bool success, ) = payable(withdrawTo).call{value: intent.rewardNative}("");
+         require(success, "ETH transfer failed");
-         emit Withdrawal(_hash, withdrawTo);
      }
  }

  function _fulfill(
      uint256 _sourceChainID,
      address[] calldata _targets,
      bytes[] calldata _data,
      uint256 _expiryTime,
      bytes32 _nonce,
      address _claimant,
      bytes32 _expectedHash
  ) internal validated(_expiryTime, msg.sender) returns (bytes[] memory) {
      bytes32 intentHash = encodeHash(_sourceChainID, block.chainid, address(this), _targets, _data,
↪   _expiryTime, _nonce);
+     // Checks
      if (intentHash != _expectedHash) revert InvalidHash(_expectedHash);
      if (fulfilled[intentHash] != address(0)) revert IntentAlreadyFulfilled(intentHash);

+     // Effects
+     fulfilled[intentHash] = _claimant;
+     emit Fulfillment(_expectedHash, _sourceChainID, _claimant);

      // Store the results of the calls
      bytes[] memory results = new bytes[](_data.length);

+     // Interactions
      for (uint256 i = 0; i < _data.length; i++) {
          address target = _targets[i];
          if (target == mailbox) revert CallToMailbox();
          (bool success, bytes memory result) = _targets[i].call(_data[i]);
          if (!success) revert IntentCallFailed(_targets[i], _data[i], result);
          results[i] = result;
      }

-     fulfilled[intentHash] = _claimant;
-     emit Fulfillment(_expectedHash, _sourceChainID, _claimant);

      return results;
  }
```

4. Use .call() for ETH Transfers: Replace `.transfer()` with `.call()` for ETH transfers to prevent potential failures with complex receiving contracts due to gas limitations (in `drain`, `_fulfill`, `withdrawRewards`, `batchWithdraw`):

```diff
  function drain(address _destination) public onlyOwner {
-     payable(_destination).transfer(address(this).balance);
+     (bool success, ) = payable(_destination).call{value: address(this).balance}("");
+     if (!success) revert NativeTransferFailed();
  }
```

5. Support Native Token Transfers and Gas Limits in Target Calls: Currently, native token transfers must be executed through the `transferNative` function, and there's no way to specify gas limits for individual target calls. The protocol could be improved by allowing direct value transfers and gas limits in target calls:

```diff
  // In Intent.sol
  struct Intent {
      address creator;
      uint256 destinationChainID;
      address[] targets;
      bytes[] data;
+     uint256[] values;  // Amount of native token to send with each call
+     uint256[] gasLimits;  // Gas limit for each target call
      address[] rewardTokens;
      uint256[] rewardAmounts;
      uint256 expiryTime;
      bool hasBeenWithdrawn;
      bytes32 nonce;
      address prover;
      uint256 rewardNative;
  }

  // In IntentSource.sol
  function createIntent(
      uint256 _destinationChainID,
      address _inbox,
      address[] calldata _targets,
      bytes[] calldata _data,
+     uint256[] calldata _values,
+     uint256[] calldata _gasLimits,
      address[] calldata _rewardTokens,
      uint256[] calldata _rewardAmounts,
      uint256 _expiryTime,
      address _prover
  ) external payable returns (bytes32 intentHash) {
      // ...
+     if (_values.length != _targets.length) revert ValuesMismatch();
+     if (_gasLimits.length != _targets.length) revert GasLimitsMismatch();
      bytes32 intermediateHash = keccak256(abi.encode(
          CHAIN_ID,
          _destinationChainID,
          _targets,
          _data,
+         _values,
+         _gasLimits,
          _expiryTime,
          _nonce
      ));
      // ... rest of function
  }

  // In Inbox.sol
  function _fulfill(
      uint256 _sourceChainID,
      address[] calldata _targets,
      bytes[] calldata _data,
+     uint256[] calldata _values,
+     uint256[] calldata _gasLimits,
      uint256 _expiryTime,
      bytes32 _nonce,
      address _claimant,
      bytes32 _expectedHash
  ) internal validated(_expiryTime, msg.sender) returns (bytes[] memory) {
      // ... hash validation and effects ...

      bytes[] memory results = new bytes[](_data.length);
      for (uint256 i = 0; i < _data.length; i++) {
          address target = _targets[i];
```

```
            if (target == mailbox) revert CallToMailbox();
-           (bool success, bytes memory result) = _targets[i].call{value: _values[i]}(_data[i]);
+           if (gasleft() < _gasLimits[i] + 100) revert InsufficientGasLeft();
+           (bool success, bytes memory result) = _targets[i].call{
+               value: _values[i],
+               gas: _gasLimits[i]
+           }(_data[i]);
            if (!success) revert IntentCallFailed(_targets[i], _data[i], result);
            results[i] = result;
        }

        return results;
    }
```

6. Consider using `Ownable2Step` instead of `Ownable`: Use of `Ownable` for `Inbox` allows a single-step ownership transfer which is risky because it does not protect against accidental transfers to incorrect/invalid addresses. `Ownable2Step` enforces a two-step transfer which protects against such errors.

Eco: Partially fixed in PR 111. (6.3) and (6.4) were fixed here in PR 123. (6.2) and (6.6) are acknowledged. For (6.5) we're planning to do it in our next release.

Cantina Managed: Reviewed that the aforementioned PRs implement part of the protocol design improvement recommendations.

3.6.3 Code quality and documentation improvements

Severity: Informational

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

Description/Recommendation:

1. Remove Commented-Out Declarations: The codebase contains commented-out code declarations that add noise without providing value. These should be removed for better code readability:

```
    contract Prover is SimpleProver {
-       // uint16 public constant NONCE_PACKING = 1;
        ProofType public constant PROOF_TYPE = ProofType.Storage;
```

2. Complete Natspec Documentation and Remove TODOs: The codebase contains incomplete documentation marked with "todo" placeholders. These should be properly documented:

```
    /**
    * @notice Validates L2 world state by ensuring that the passed in l2 world state root corresponds to value
↪   in the L2 output oracle on L1
    * @param claimant the address that can claim the reward
    * @param inboxContract the address of the inbox contract
    * @param intermediateHash the hash which, when hashed with the correct inbox contract, will result in the
↪   correct intentHash
-    * @param l2StorageProof todo
-    * @param rlpEncodedInboxData todo
-    * @param l2AccountProof todo
-    * @param l2WorldStateRoot todo
+    * @param l2StorageProof Merkle proof for the storage slot in the inbox contract
+    * @param rlpEncodedInboxData RLP encoded account data of the inbox contract
+    * @param l2AccountProof Merkle proof for the inbox contract account
+    * @param l2WorldStateRoot The state root of the L2 chain being proven
    */
    function proveIntent(
        uint256 chainId,
        address claimant,
        address inboxContract,
        bytes32 intermediateHash,
        bytes[] calldata l2StorageProof,
        bytes calldata rlpEncodedInboxData,
        bytes[] calldata l2AccountProof,
        bytes32 l2WorldStateRoot
    ) public {
```

3. Remove Unused Configuration Fields:

```
struct ChainConfiguration {
-   uint8 provingMechanism;
    uint256 settlementChainId;
    address settlementContract;
    address blockhashOracle;
-   uint256 outputRootVersionNumber;
}
```

4. Change Function Visibility to External:

```
- function rlpEncodeDataLibList(bytes[] memory dataList) public pure returns (bytes memory) {
+ function rlpEncodeDataLibList(bytes[] memory dataList) external pure returns (bytes memory) {
```

5. Fix Incorrect Documentation:

```
- * @param rlpEncodedBlockData properly encoded L1 block data
+ * @param rlpEncodedBlockData properly encoded L2 block data
```

6. Add missing `NATSPEC`:

```
  * @param _destinationChainID the destination chain
+  * @param _inbox the address of the inbox contract
    * @param _targets the addresses on _destinationChainID at which the instructions need to be executed
    * @param _data the instruction sets to be executed on _targets
    * @param _rewardTokens the addresses of reward tokens
    * @param _rewardAmounts the amounts of reward tokens
    * @param _expiryTime the timestamp at which the intent expires
    * @param _prover the prover against which the intent's status will be checked
    */
    function createIntent(
        uint256 _destinationChainID,
        address _inbox,
        address[] calldata _targets,
        bytes[] calldata _data,
        address[] calldata _rewardTokens,
        uint256[] calldata _rewardAmounts,
        uint256 _expiryTime,
        address _prover
```

7. Remove comment: The following line is commented out, remove it:

```
  } else {
            gameStatusStorageSlotRLP = bytes.concat(
                RLPWriter.writeBytes(
                    abi.encodePacked(
-                        // abi.encodePacked(l2BlockNumberChallenged),
                        abi.encodePacked(initialized),
                        abi.encodePacked(gameStatus),
                        abi.encodePacked(resolvedAt),
                        abi.encodePacked(createdAt)
                    )
                )
            );
```

Eco: Fixed in PR 110. Partially fixed (8.6) in PR 111. Specifically, the following commits address:

1. Remove commented out Declarations: 301487bc.
2. Complete Natspec Documentation and Remove TODOs: 0d191771, also updated Natspec for `proveWorldStateBedrock` 0d191771.
3. Remove Unused Configuration Fields: *leaving for now -- as they will be used/updated in future release see ED-4357*.
4. Change Function Visibility to External: addressed in e7b9643e.
5. Fix Incorrect Documentation: Not sure what line this is referring to, but I believe the rlpEncodedBlockData is from the L1 (or settlement) Block.
6. Add missing `NATSPEC`: For `createIntent` addressed in 8997382b.

Cantina Managed: Reviewed that the changes in the PRs implement the code quality and documentation recommendations.

3.6.4 Malicious intent creator can provide an invalid Inbox contract to grief solvers

Severity: Informational

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

Description: A malicious intent creator can provide an invalid Inbox contract, while creating the intent in `createIntent()`, that griefs solvers when they attempt to fulfill that intent on the protocol-deployed valid Inbox contract on destination chain.

An intent creator is expected to create intents on the source chain via a call to `createIntent()`, which includes the `_inbox` address of the protocol-deployed valid Inbox contract on destination chain. This `_inbox` address is included in the `intentHash`. Solvers tracking intent creation via `IntentCreated` event emits are expected to execute these intents on the protocol-deployed valid Inbox contracts on appropriate destination chains as specified by the intent creators' `destinationChainIDs`.

However, there is no validation in `createIntent()` to check that the specified `_inbox` contract is indeed the protocol-deployed valid Inbox contract on the destination chain. Solvers attempting to execute intents created with invalid Inbox contracts will revert in `Inbox._fulfill()` because of mismatched intent hashes.

Impact: Low, because intent creators can only grief solvers by forcing their attempted intent fulfillment to revert on destination chains. Solvers may be overwhelmed with such spam intents.

Likelihood: Very Low, because intent creators need to be motivated to grief solvers and spam the system with such invalid intents. Per protocol assumption, solvers are expected to be sophisticated entities which will validate intents before attempting to fulfill them.

Recommendation:

1. Consider including the `_inbox` address explicitly in the `IntentCreated` event emitted in `createIntent()` so that solvers can easily check if that matches the protocol-deployed valid Inbox contract on the destination chain. Currently solvers have to check this by validating `intentHash`.
2. Consider enforcing a check in `createIntent()` to validate that the specified `_inbox` contract is indeed the protocol-deployed valid Inbox contract on the destination chain.

Eco: Acknowledged. Right now our inbox contract is the only one, and even in the case that someone tries to solve an intent on a different inbox, it will fail due to the hashes mismatching.

Cantina Managed: Acknowledged.

3.6.5 Privileged functions are missing event emissions

Severity: Informational

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

Privileged `onlyOwner` Inbox functions `setMailbox()` and `drain()` are missing event emissions. Emitting events when the Hyperlane mailbox is initialized during deployment and when reclaiming native tokens from dust and excess fees will help improve transparency and monitoring.

Recommendation: Consider emitting events in `setMailbox()` and `drain()`.

Eco: Fixed in PR 119.

Cantina Managed: Reviewed that PR 119 fixes the issue as recommended. `drain()` has been removed.

3.6.6 Malicious solvers can grief other solvers by setting `_claimant` to zero address

Severity: Informational

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

Description: Solvers are expected to set a valid `_claimant` address in `_fulfill()` with `fulfilled[intentHash] = _claimant` so that the claimant can withdraw rewards on the source chain after proving intent fulfillment.

A solver Alice may execute an intent's target calls but however set `_claimant` to zero address thereby forfeiting any intent rewards. This will allow another solver Bob who tries to fulfill the same intent (perhaps initiated around the same time as Alice but was sequenced after Alice) to pass the `fulfilled[intentHash] != address(0)` check for `IntentAlreadyFulfilled` but later possibly revert in one of the intent's target calls because that is being executed for a second time (Alice executed it the first time). Alice manages to grief Bob at the expense of forfeiting intent rewards.

Recommendation: Consider adding a `_claimant != address(0)` check in `_fulfill()` because forfeiting/burning any intent rewards should not be intentional and likely a solver error.

Eco: Fixed in PR 124.

Cantina Managed: Reviewed that PR 124 fixes the issue as recommended.

3.6.7 Missing event emit for `solverWhitelist` initialization

Severity: Informational

Context: Inbox.sol#L54

Description: When emitting events, state updates need to be tracked from the first initialization. The goal is to reach the initial state if you would look back at all the logs from a specific event. In this case, `solverWhitelist` is not tracked properly by emitting its corresponding event:

```
constructor(address _owner, bool _isSolvingPublic, address[] memory _solvers) Ownable(_owner){
    isSolvingPublic = _isSolvingPublic;
    for (uint256 i = 0; i < _solvers.length; i++) {
        solverWhitelist[_solvers[i]] = true;
    }
}
```

Given that there exists a specific event `SolverWhitelistChanged` to track its storage updates as used in `changeSolverWhitelist()`:

```
function changeSolverWhitelist(address _solver, bool _canSolve) public onlyOwner {
    solverWhitelist[_solver] = _canSolve;
    emit SolverWhitelistChanged(_solver, _canSolve);
}
```

Recommendation: Consider tracking the updates to each whitelisted solver accordingly:

```
  constructor(address _owner, bool _isSolvingPublic, address[] memory _solvers) Ownable(_owner){
      isSolvingPublic = _isSolvingPublic;
      for (uint256 i = 0; i < _solvers.length; i++) {
          solverWhitelist[_solvers[i]] = true;
+         emit SolverWhitelistChanged(_solvers[i], true);
      }
  }
```

Eco: Fixed in PR 119.

Cantina Managed: Reviewed that PR 119 resolves the issue as recommended.

4 Appendix

4.1 Proof security improvements

1. Limit Parameters: Add a reasonable upper bound to prevent arbitrary storage slot targeting:

```
  function proveWorldStateBedrock(
      uint256 chainId,
      bytes calldata rlpEncodedBlockData,
      bytes32 l2WorldStateRoot,
      bytes32 l2MessagePasserStateRoot,
      uint256 l2OutputIndex,
      bytes[] calldata l1StorageProof,
      bytes calldata rlpEncodedOutputOracleData,
      bytes[] calldata l1AccountProof,
      bytes32 l1WorldStateRoot
  ) public virtual {
+     if (l2OutputIndex > 2**64) revert OutputIndexTooHigh();
      // ... rest of function
  }
```

2. Minimize User-Provided Proof Parameters: Derive values internally rather than accepting untrusted input. When using user input SOLELY for additional validation clearly name the variables as such. These should not be used to directly set state when they can also be derived:

```
- bytes32 l1WorldStateRoot,
+ bytes32 expectedL1WorldStateRoot,
  // ...
+ bytes32 provenl1WorldStateRoot = derivedL1WorldStateRoot();
+ require(provenl1WorldStateRoot == expectedL1WorldStateRoot, "Root mismatch");
```

3. Separate Proof Logic: Split the Cannon proof verification into distinct stages. E.g. the first part of the cannon proof in `_faultDisputeGameFromFactory` uses `disputeGameFactoryProofData` in order to calculate `rootClaim` which is only ever used in the second part in `faultDisputeGameIsResolved`.

4. Improve Variable Naming Consistency: Use clear and consistent naming conventions for all proof-related variables:

```
  function proveWorldStateBedrock(
-     uint256 chainId,
+     uint256 l2ChainId,
-     bytes calldata rlpEncodedBlockData,
+     bytes calldata l2RlpBlockData,
      bytes32 l2WorldStateRoot,
-     bytes32 messagePasserStateRoot,
+     bytes32 l2MessagePasserStateRoot,
      uint256 l2OutputIndex,
-     bytes[] calldata l1StorageProof,
+     bytes[] calldata l1OutputOracleStorageProof,
-     bytes calldata rlpEncodedOutputOracleData,
+     bytes calldata rlpOutputOracleAccountData,
-     bytes32 l1AccountProof,
+     bytes32 l1OutputOracleAccountProof,
      bytes32 l1WorldStateRoot
  ) public {
      // ...
-     bytes memory outputOracleStateRoot
+     bytes memory outputOracleStorageRoot
  }
```

The naming should:

- Clearly indicate the type of root (storage vs. state).
- Include the layer prefix (L1/L2).
- Specify the contract when applicable (`MessagePasser`, `OutputOracle`).
- Use consistent terminology across the codebase.

5. Add Fuzz Testing: Add property-based tests focusing on proof parameter validation:

```
function testFuzz_proveWorldStateBedrock(uint256 l2OutputIndex) public {
    // Set all other parameters to fixed values
    vm.assume(l2OutputIndex != L2_OUTPUT_INDEX);
    vm.expectRevert();
    proveWorldStateBedrock(
        CHAIN_ID,
        ENCODED_BLOCK_DATA,
        L2_STATE_ROOT,
        MSG_PASSER_ROOT,
        l2OutputIndex,  // Fuzzed parameter
        L1_STORAGE_PROOF,
        ENCODED_ORACLE_DATA,
        L1_ACCOUNT_PROOF,
        L1_STATE_ROOT
    );
}
```