Polymer
Security Review

Cantina Managed review by:

Mario Poneder, Security Researcher
Frank Castle, Security Researcher

June 25, 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

Polymer makes interoperability fast and easy for application builders.

From Jun 2nd to Jun 10th the Cantina team conducted a review of polymer-monorepo on commit hash 8b8a3c34. The team identified a total of **16** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	4	4	0
Medium Risk	2	2	0
Low Risk	6	6	0
Gas Optimizations	0	0	0
Informational	4	4	0
Total	**16**	**16**	**0**

3 Findings

3.1 High Risk

3.1.1 Validation results may be truncated or rejected due to return data and log size limits

Severity: High Risk

Context: lib.rs#L220-L229

Description: The `polymer_prover` program returns validation results using `set_return_data` in the CPI case, which is strictly limited to 1024 bytes by `MAX_RETURN_DATA`. If the `ValidateEventResult::Valid` event exceeds this size, the Solana runtime will reject the return data with a `ReturnDataTooLarge` error, causing the CPI call to fail and preventing the caller from receiving any result. In the non-CPI case, results are emitted using `emit!`, which is limited by the `LOG_MESSAGES_BYTES_LIMIT` of 10kB per transaction, and this limit is shared with all other logs and events in the transaction.

In both cases, if the `ValidateEventResult::Valid` event is too large, it is either silently truncated (non-CPI) or causes an error (CPI), which can result in clients receiving incomplete or unusable validation results. This is especially problematic for CPI integrations that rely on successful invocation, impeding the main use case of the program.

Recommendation: It is recommended to store the validation result in a dedicated account instead of relying on return data or logs. This approach avoids silent truncation, ensures reliable delivery of large results, and allows clients to fetch the full validation output regardless of transaction log or return data limits.

Polymer: Fixed in PR 24.

Cantina Managed: Fix verified.

3.1.2 Missing Verification of the L2 World State Root in the `proveL2Native` and `proveSettledState` Functions

Severity: High Risk

Context: OPStackBedrockProver.sol#L142

Description: In the `proveL2Native` function, the storage slot to be verified is validated against the L2 world state root, which in turn is expected to be verified through a call to the Bedrock prover.

However, in the Bedrock prover, the output root is verified against the oracle account's storage root. The function responsible for generating this output root receives the L1 world state root instead of the L2 world state root, which results in an incorrect output root and invalid verification.

```
outputRoot = _generateOutputRoot(
    _chainConfig.versionNumber, _args._l1WorldStateRoot, _args._l2MessagePasserStateRoot, blockHash
);

function _generateOutputRoot(
    uint256 _outputRootVersion,        // Version
    bytes32 _worldStateRoot,           // L1 state root (incorrect)
    bytes32 _messagePasserStateRoot,   // Message passer root
    bytes32 _latestBlockHash           // L2 block hash
) internal pure returns (bytes32) {
    return keccak256(abi.encode(_outputRootVersion, _worldStateRoot, _messagePasserStateRoot,
    ↪  _latestBlockHash));
}
```

Impact: If `_l2WorldStateRoot` is not included in the output root computation (due to the incorrect use of `_args._l1WorldStateRoot`), the function does not verify that `_l2WorldStateRoot` matches the state root recorded in the `L2OutputOracle`.

This allows an attacker to supply an arbitrary `_l2WorldStateRoot` that is not checked against the L1-settled L2 state. Consequently, the trustlessness of the verification process is undermined, potentially resulting in validation against a fake or manipulated L2 state root. This has a **high impact** as it compromises the correctness of proofs.

Recommendation: To address this issue and ensure the `_l2WorldStateRoot` is properly verified:

1. Fix the `_generateOutputRoot` Call: Update the `_proveWorldStateBedrock` function to use `_args._l2WorldStateRoot` instead of `_args._l1WorldStateRoot`:

   ```
   bytes32 outputRoot = _generateOutputRoot(
       _chainConfig.versionNumber,
       _args._l2WorldStateRoot,
       _args._l2MessagePasserStateRoot,
       keccak256(_rlpEncodedL2Header)
   );
   ```

2. Verify the L2 Header State Root: Extract the state root from `_rlpEncodedL2Header` (field 3) and compare it with `_args._l2WorldStateRoot` to ensure consistency:

   ```
   bytes32 l2HeaderStateRoot = bytes32(RLPReader.readBytes(RLPReader.readList(_rlpEncodedL2Header)[3]));
   if (l2HeaderStateRoot != _args._l2WorldStateRoot) {
       revert InvalidL2StateRoot(l2HeaderStateRoot, _args._l2WorldStateRoot);
   }
   ```

   This step ensures that `_l2WorldStateRoot` accurately reflects the actual L2 block header's state root, maintaining trust in the verification process.

Polymer: Fixed in PR 146.

Cantina Managed: Fix verified.

3.1.3 Missing Validation for `faultDisputeGameStateRoot` Allows invalid Finality

Severity: High Risk

Context: OPStackCannonProver.sol#L212, OPStackCannonProver.sol#L229

Description: The `faultDisputeGameStateRoot` is not validated against any trusted parameters such as the `_l1WorldStateRoot`. Currently, only the `rlpEncodedFaultDisputeGameData` is validated. As a result, an attacker can pass any arbitrary, malicious, or invalid `faultDisputeGameStateRoot` in the proof process. This unvalidated root is used in the following locations:

```
// Verify game status storage proof
ProverHelpers.proveStorageBytes32(
    abi.encodePacked(_faultDisputeGameStatusSlot),
    faultDisputeGameStatusStorage,
    _faultDisputeGameProofData.faultDisputeGameStatusStorageProof,
    bytes32(_faultDisputeGameProofData.faultDisputeGameStateRoot)
);
```

```
// Prove that the FaultDispute game has been settled
ProverHelpers.proveStorageBytes32(
    abi.encodePacked(_faultDisputeGameRootClaimSlot),
    _rootClaim,
    _faultDisputeGameProofData.faultDisputeGameRootClaimStorageProof,
    bytes32(_faultDisputeGameProofData.faultDisputeGameStateRoot)
);
```

Without verifying that the `faultDisputeGameStateRoot` is derived from or included in the trusted `rlpEncodedFaultDisputeGameData`, the process can falsely conclude that a Fault Dispute Game has been resolved-even when it has not. This breaks the resolution constraint.

Impact: This allows an attacker to finalize a fraudulent or disputed L2 state, effectively bypassing the finality rules. Disputed games may be prematurely accepted as successful or failed, allowing invalid state transitions to be considered finalized.

Recommendation: Introduce a validation step that ensures the `faultDisputeGameStateRoot` is verifiably included in or derived from the `rlpEncodedFaultDisputeGameData`. This guards against tampering or spoofing and ensures only legitimate state roots are accepted during the proof process.

Polymer: Fixed in PR 147.

Cantina Managed: Fix verified.

3.1.4 `NativeProver` allows proofs of stale L2 states leading to potential false positives and negatives

Severity: High Risk

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

Description: It is assumed that proofs of the `NativeProver` contract from the fallback path always prove the latest state on the counterparty L2 chain, since they require a recent view of the L1 block hash on the local L2. However, this assumption is incorrect. In practice, all previously settled L2 states are retained on L1 (for both OP Stack Cannon and Bedrock), and proofs can be constructed for any of these historical states. This leads to two main issues:

1. False Positives: A user can prove that a state `S` exists on chain A (counterparty) to chain B (local), even if `S` is no longer current. By using a proof referencing an older L2 output or dispute game factory proxy (which still exists on the most recent L1 block), the user can convince chain B that `S` exists, even though it has been updated to `S'` in a more recent block.
2. False Negatives: If a state changes multiple times between L1 settlements (e.g., `S` → `S'` → `S`), and `S'` does not exist in any block that was settled on L1, it cannot be proven at all. This means some valid states may be unprovable if they do not coincide with a settled output.

This behavior can be problematic for DApps or protocols that assume a proven state is always the latest or canonical state.

Recommendation: It is recommended to implemented the following mitigation measures:

- Acknowledge and document the limitation: Clearly document that the protocol only proves existence of a state at some point in the past, not necessarily the latest state. DApp developers should be advised to only rely on states that are monotonic or irrevocable, or to locally track and reject stale states.
- Return proven block number: Update the affected methods to return the block number associated with the proven state. This allows DApps to check the recency of the proof and enforce their own freshness/staleness policies.

Disclaimer: This protocol-level limitation / potential vulnerability was discovered by the Polymer team and included in the report for the sake of completeness..

Polymer: Fixed in PR 148.

Cantina Managed: Fix verified.

3.2 Medium Risk

3.2.1 Possible revocation of Irrevocable Roles in `Registry.sol`

Severity: Medium Risk

Context: Registry.sol#L95-L101

Description: The `Registry` contract allows revoking roles for irrevocable chain IDs because it does not override the `revokeRole` function from OpenZeppelin's `AccessControl` library to enforce irrevocability. This creates a vulnerability where an admin can revoke an irrevocable role, potentially leaving an irrevocable chain without any grantees. Such a scenario could enable a DoS by preventing configuration updates for that chain, as no grantee would be authorized to manage it.

```
function grantChainIDIrrevocable(address _grantee, uint256 _chainID) external onlyOwner isRevocable(_chainID) {
    return _grantChainIDIrrevocable(_grantee, _chainID);
}

function _grantChainIDIrrevocable(address _grantee, uint256 _chainID) internal isRevocable(_chainID) {
    BitMaps.set(_irrevocableChainIDBitmap, _chainID);
    bytes32 role = _getChainRole(_chainID);
    _grantRole(role, _grantee);
    emit NewIrrevocableGrantee(_chainID, _grantee);
}
```

The `Registry` contract uses a `BitMaps.BitMap (_irrevocableChainIDBitmap)` to mark chain IDs as irrevocable, preventing further grants or modifications via the `isRevocable` modifier. However, the `revokeRole` function inherited from `AccessControl` does not check `_irrevocableChainIDBitmap`, allowing an admin (`DEFAULT_ADMIN_ROLE`) to revoke roles for irrevocable chain IDs.

```
function revokeRole(
    bytes32 role,
    address account
) public virtual override onlyRole(getRoleAdmin(role)) {
    _revokeRole(role, account);
}
```

Recommendation: To mitigate this, the `revokeRole` function should be overridden to only permit revocation for revocable roles on revocable chain IDs, ensuring that irrevocable roles remain protected and chains always have at least one authorized grantee.

Polymer: Fixed in PR 145.

Cantina Managed: Fix verified.

3.2.2 Inconsistent role assignment behavior for chain ID grantees in `Registry` contract

Severity: Medium Risk

Context: Registry.sol#L95-L101

Description: The current implementation of the `Registry` contract's internal role-granting functions allows for inconsistent grantee management for chain IDs:

- `_grantChainIDIrrevocable` is intended to allow only a single grantee for a given chain ID by using the `isRevocable` modifier, which prevents further changes once set.
- `_grantChainID` can assign the same role to multiple grantees for a given chain ID, as expected for revocable roles.
- There is no mechanism in `_grantChainIDIrrevocable` to revoke roles from previous grantees that were assigned via `_grantChainID`, leading to a situation where multiple addresses may retain permissions for a chain ID that should be irrevocably assigned to only one grantee.
- Additionally, the inherited `grantRole` function from OpenZeppelin's `AccessControl` contract can be called directly, bypassing the intended access control logic and allowing arbitrary assignment of roles without regard to the revocability or uniqueness constraints.

This behavior is inconsistent with the intended design, where an irrevocable chain ID should have only one grantee, and no further changes should be possible.

Recommendation: It is recommended to apply the following mitigation measures:

- Update `_grantChainIDIrrevocable` to revoke roles from all previous grantees for the given chain ID before assigning the new irrevocable grantee.
- Consider overriding the inherited `grantRole` function to prevent bypassing the intended role assignment logic.

Polymer: Fixed in PR 150. The patched behavior is now that instead of automatically removing all the revocable roles when an irrevocable role is added, we disallow adding any new roles (revocable or irrevocable), but still allow revoking the revocable roles.

Cantina Managed: Fix verified.

3.3 Low Risk

3.3.1 Fixed-Size Reallocation Prevents Resizing of Proof Cache Account

Severity: Low Risk

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

Description: The function `resize_proof_cache` does not actually resize the proof cache account, as it uses a fixed constant size for the account:

```
#[derive(Accounts)]
pub struct ResizeProofCache<'info> {
    #[account(
        mut,
        realloc = DISCRIMINATOR_SIZE + ProofCacheAccount::INIT_SPACE,
        realloc::payer = authority,
        realloc::zero = false,
        seeds = [authority.key().as_ref()],
        bump,
    )]
    pub cache_account: Account<'info, ProofCacheAccount>,
    #[account(mut, signer)]
    pub authority: Signer<'info>,
    // Required to mutate the PDA account
    pub system_program: Program<'info, System>,
}
```

Recommendation: Use a variable length to dynamically extend the account:

```
#[derive(Accounts)]
#[instruction(input: String)]
pub struct ResizeProofCache<'info> {
    #[account(
        mut,
        realloc = DISCRIMINATOR_SIZE + ProofCacheAccount::INIT_SPACE + input.len(),
        realloc::payer = authority,
        realloc::zero = false,
        seeds = [authority.key().as_ref()],
        bump,
    )]
    pub cache_account: Account<'info, ProofCacheAccount>,
    #[account(mut, signer)]
    pub authority: Signer<'info>,
    // Required to mutate the PDA account
    pub system_program: Program<'info, System>,
}
```

This allows the account size to be adjusted based on the length of the input.

Polymer: Fixed in PR 16.

Cantina Managed: Fix verified.

3.3.2 `secp256k1_recover` in `recover_signature` is susceptible to signature malleability

Severity: Low Risk

Context: validate_event.rs#L144-L151

Description: The `recover_signature` function uses `secp256k1_recover` to recover Ethereum addresses from signatures. However, this approach is susceptible to signature malleability, as it does not enforce the low-S value requirement for ECDSA signatures. Without restricting to low-S signatures, one could produce two valid signatures for the same message.

Recommendation: It is recommended to enforce a low-S value check on ECDSA signatures before calling `secp256k1_recover` to prevent signature malleability, ensuring only canonical signatures are accepted.

Polymer: Fixed in PR 18.

Cantina Managed: Fix verified.

3.3.3 Initialization frontrunning vulnerability in `polymer_prover` due to permissionless `initialize` instruction

Severity: Low Risk

Context: lib.rs#L154-L177

Description: The `initialize` instruction in the `polymer_prover` program is currently permissionless, allowing any user to invoke it immediately after deployment. This enables the first caller to set the `authority`, `client_type`, `signer_addr`, and `peptide_chain_id` fields of the `internal` account, potentially locking out the intended owner or misconfiguring critical parameters. This exposes the program to frontrunning attacks where a malicious actor can take control of the initialization process.

Recommendation: It is recommended to implement access control in the `initialize` instruction to restrict who can perform initialization, for example by requiring the instruction to be co-signed by the keypair of program which should only be known to the deployer:

```
+ // 2 signatures
  #[account(mut, signer)]
  pub authority: Signer<'info>,
+ #[account(address = crate::ID)]
+ pub program: Signer<'info>,
```

Polymer: Fixed in PR 23.

Cantina Managed: Fix verified.

3.3.4 Unused error definitions indicate missing validation logic in prover contracts

Severity: Low Risk

Context: NativeProver.sol#L63-L71, NativeProver.sol#L78-L81, NativeProver.sol#L93-L96, OPStackBedrockProver.sol#L52-L55, OPStackCannonProver.sol#L69-L72, CrossL2ProverV2.sol#L31-L32

Description: Several custom error types are defined in the `NativeProver`, `OPStackBedrockProver`, `OPStackCannonProver`, and `CrossL2ProverV2` contracts, but are never used in the code. This suggests that important validation steps may be missing, and certain failure cases are not being checked or reported, specifically:

- `NeedLaterBlock` and `OutdatedBlock` (in `NativeProver`).
- `DestinationChainStateRootNotProved` (in `NativeProver`).
- `InvalidStorageProof` (in `NativeProver`, `OPStackBedrockProver`, and `OPStackCannonProver`).
- `InvalidProofKey` and `InvalidProofValue` (in `CrossL2ProverV2`).

Recommendation: It is recommended to review the intended validation logic for each unused error and implement the corresponding checks in the contract code, or remove the error definition if not needed.

Polymer: Fixed in PR 141.

Cantina Managed: Fix verified.

3.3.5 Unused events `L1WorldStateProven` and `L2WorldStateProven` in `NativeProver`

Severity: Low Risk

Context: NativeProver.sol#L57-L61

Description: The `NativeProver` contract defines two events, `L1WorldStateProven` and `L2WorldStateProven`, but neither event is emitted anywhere in the contract. This suggests that important state changes or proof milestones are not being logged, which can hinder monitoring, debugging, and off-chain integrations.

Recommendation: It is recommended to review the intended use of `L1WorldStateProven` and `L2WorldStateProven`:

- If these events are meant to signal successful proof or state updates, add `emit` statements at the appropriate locations in the contract.
- If the events are not needed, remove their definitions to keep the codebase clean and maintainable.

Polymer: Fixed in PR 143.

Cantina Managed: Fix verified.

3.3.6 The `verifyMembership` function does not adhere to the proof layout and lacks key/value validation

Severity: Low Risk

Context: CrossL2ProverV2.sol#L196-L225

Description: The `verifyMembership` function does not fully implement the proof layout described in its comment. Specifically:

- The function only uses `proof[0]` (number of paths) and `proof[1]` (`path0start`), ignoring other header fields such as `key start`, `key end`, `value start`, and `value end`.
- It does not extract or check the key and value from the proof itself, but instead uses the input `key` and `value` parameters directly. This means the function cannot detect if the provided key or value does not match the actual data in the proof. The related custom errors `InvalidProofKey` and `InvalidProofValue` are defined but never used.
- The leaf node construction skips varint parsing and assumes a fixed layout, which may not be compatible with all proof formats.
- Path node parsing assumes static offsets and may not work for variable-length prefixes or suffixes.
- The offset update logic (`offset = offset + suffixend`) is not robust for all proof shapes and could lead to incorrect parsing.
- As a result, this function is not a general verifier and only works for a specific, simplified proof format. However, there is no comment indicating that the function is expected to be a general verifier.

Recommendation: It is recommended to revise the function concerning parsing and validation to match the comment.

Polymer: Fixed in PR 134.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 Unnecessary `init_if_needed` Check in `load_proof` Increases Compute Costs

Severity: Informational

Context: lib.rs#L90

Description: Since the cache account cannot be closed , the use of `init_if_needed` in `load_proof` results in an unnecessary check to validate whether the account already exists. This leads to increased CUs consumption and degraded performance.

```
pub struct LoadProof<'info> {
    #[account(
        init_if_needed,
        seeds = [authority.key().as_ref()],
        bump,
        payer = authority,
        space = DISCRIMINATOR_SIZE + ProofCacheAccount::INIT_SPACE,
    )]
    pub cache_account: Account<'info, ProofCacheAccount>,
    #[account(mut, signer)]
    // User will be the owner of the PDA account
    pub authority: Signer<'info>,
    // Needed to create the PDA account
    pub system_program: Program<'info, System>,
}
```

Recommendation: Create a separate instruction to initialize the cache account once. Then, remove the `init_if_needed` constraint from the `load_proof` function to eliminate the repeated existence check and improve performance.

Polymer: Fixed in PR 22.

Cantina Managed: Fix verified.

3.4.2 Unused `handler` function in `initialize.rs`

Severity: Informational

Context: initialize.rs#L3-L10

Description: The `handler` function in `initialize.rs` is currently defined but never called or referenced anywhere in the codebase. This results in dead code, which can cause confusion and may indicate incomplete implementation or refactoring.

Recommendation: It is recommended to remove the unused `handler` function if it is not needed, or ensure it is properly integrated and invoked where appropriate.

Polymer: Fixed in PR 17.

Cantina Managed: Fix verified.

3.4.3 Unused function and data structure in the `NativeProver` contract

Severity: Informational

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

Description: Unused function and data structure in the `NativeProver` contract can be removed:

```
/**
 * @notice Stores proven block data for a chain
 * @param blockNumber Number of the proven block
 * @param blockHash Hash of the proven block
 * @param stateRoot State root of the proven block
 */
struct BlockProof {
    uint256 blockNumber;
    bytes32 blockHash;
    bytes32 stateRoot;
}


function _createBlockProof(bytes32 _l2WorldStateRoot, bytes memory _rlpEncodedL2Header)
    internal
    pure
    returns (BlockProof memory blockProof)
{
    blockProof = BlockProof({
        blockNumber: _bytesToUint(RLPReader.readBytes(RLPReader.readList(_rlpEncodedL2Header)[8])),
        blockHash: keccak256(_rlpEncodedL2Header),
        stateRoot: _l2WorldStateRoot
    });
}
```

Recommendation: Remove the unused `_createBlockProof` and `BlockProof`.

Polymer: Fixed in PR 142.

Cantina Managed: Fix verified.

3.4.4 Unused struct definitions in `ReceiptParser` and `NativeProver`

Severity: Informational

Context: NativeProver.sol#L36-L39, ReceiptParser.sol#L45-L51

Description: The codebase defines the following struct types that are never used:

- `OpL2StateProof` in the `ReceiptParser` library.
- `InitialL2Configuration` in the `NativeProver` contract.

These unused structures may indicate incomplete features, leftover code from refactoring, or missing integration points. Their presence can cause confusion for maintainers and reviewers, and may increase the contract size unnecessarily.

Recommendation: It is recommended to review whether these structs are needed for future features or integrations:

- If they are not required, remove them to keep the codebase clean and maintainable.
- If they are intended for future use, add a comment explaining their purpose and planned usage.

Polymer: Fixed in PR 133.

Cantina Managed: Fix verified.