Eco Routes
Security Review

Cantina Managed review by:

0xRajeev, Lead Security Researcher
Sujith Somraaj, Security Researcher

September 4, 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 May 7th to May 10th the Cantina team conducted a review of eco-routes on commit hash 77156f0f. The team identified a total of **14** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	2	1	1
Low Risk	3	2	1
Gas Optimizations	0	0	0
Informational	7	2	5
Total	**14**	**7**	**7**

Disclaimer: any changes made on the reviewed codebase outside the scope of this engagement during the period of review or fix-review is considered out of scope and may present a significant risk.

3 Findings

3.1 High Risk

3.1.1 Incorrect `IPROVER_INTERFACE_ID` constant allows malicious solvers to steal funds during fulfillment

Severity: High Risk

Context: Inbox.sol#L29

Description: The `IPROVER_INTERFACE_ID` constant identifies prover contracts and is set with an incorrect value.

```
bytes4 public constant IPROVER_INTERFACE_ID = 0xd8e1f34f; //type(IProver).interfaceId
```

The value `0xd8e1f34f` does not correspond to what's obtained from `type(IProver).interfaceId` and is likely a stale value from earlier. As a result, `Inbox.sol` contract will fail to recognize valid prover contracts during intent fulfillment process. This allows malicious solvers to call prover contracts that should be restricted during fulfillment, thereby circumventing an essential security mitigation.

The security check in the `_fulfill()` function, which is supposed to block calls to prover contracts, would fail to do so and allow them to be called in the snipper shown below:

```
(bool isProverCall, ) = (call.target).call(
    abi.encodeWithSignature(
        "supportsInterface(bytes4)",
        IPROVER_INTERFACE_ID
    )
);
if (isProverCall) {
    // call to prover
    revert CallToProver();
}
```

Impact Explanation: High, because unauthorized calls to `prover` could lead to fund theft from solvers by maliciously setting `_claimant` addresses in the prover contract.

Likelihood Explanation: Medium, because the current implementation of identifying the prover contract is itself incorrect (see the finding "Incorrect check for prover calls may lead to DoS for intent fulfillment").

Recommendation: Consider changing the declaration of the `IPROVER_INTERFACE_ID` constant to `type(IProver).interfaceId` instead of hardcoding the value. This can be done at declaration or within the constructor.

Eco: Fixed in PR 241.

Cantina Managed: Reviewed that PR 241 replaces the hardcoded value and instead sets it with `IPROVER_INTERFACE_ID = type(IProver).interfaceId;` in the constructor as recommended.

3.1.2 Incorrect equivalence assumption on chain IDs and prover domain IDs may lead to loss of solver funds

Severity: High Risk

Context: HyperProver.sol#L297, MetaProver.sol#L251

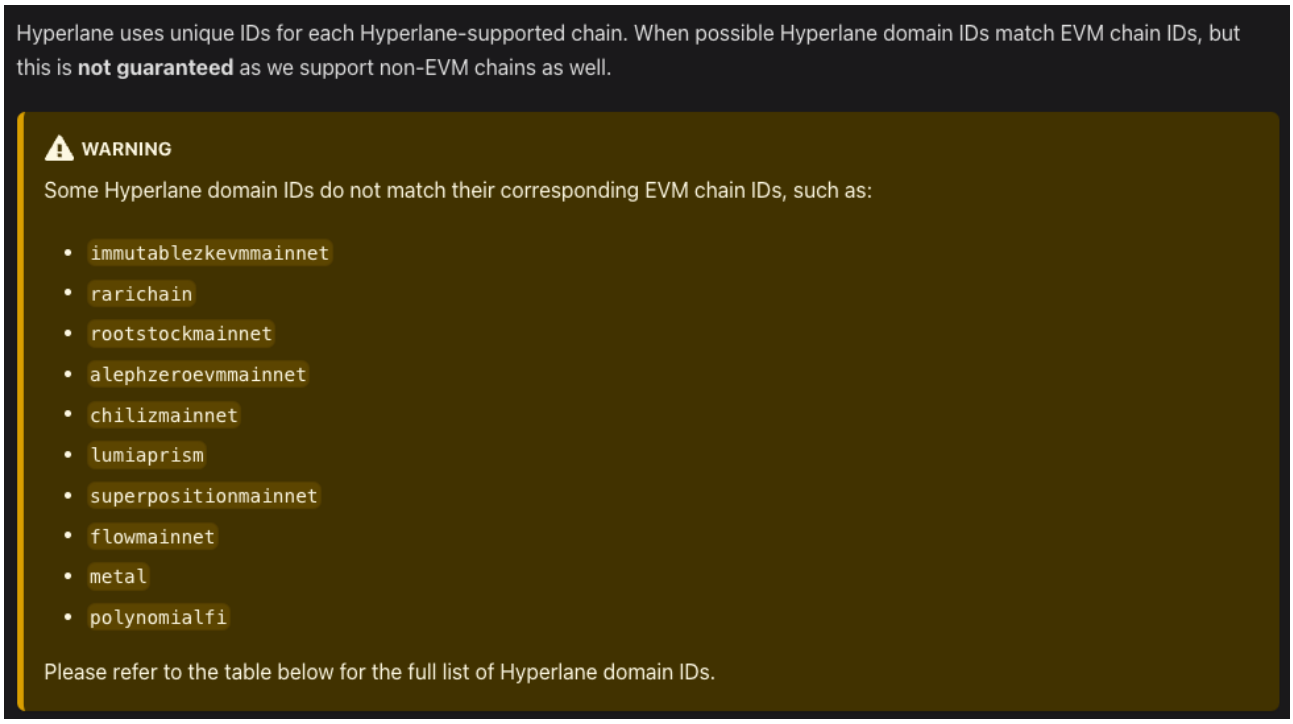
Summary: Incorrect equivalence assumption on chain IDs and prover domain IDs while formatting prover messages will lead to loss of solver funds for cases where the origin chain IDs are different from prover domain IDs.

Finding Description: Cross-chain frameworks of Hyperlane and Metalayer used by Eco protocol have a notion of domain IDs to correspond to their different supported chains. While many such chains have their domain IDs to be the same as their chain IDs, this is not always the case.

Hyperlane has an explicit warning, that says:

> Hyperlane domain IDs are not guaranteed to match chain IDs.

This warning is also specified in their documentation for Domains as shown below:



Metalayer also has a similar scenario where the equivalence is not true for all their supported chains, e.g. Curtis, as per their documentation for deployments.

However, the current implementation assumes this equivalence while formatting prover messages, as shown below in the derivation of domain from the chain ID:

```
domain = uint32(_sourceChainId);
```

For chains where this equivalence does not hold, the prover messages are sent to an incorrect chain different from the intent's source/origin chain, which leads to loss of such messages on the intended chain. As a result, a solver that has correctly fulfilled an intent on the destination will never have it's proof sent to the correct source, which thereby prevents it from claiming its deserved rewards. This effectively leads to loss of solver funds used to fulfill the intent.

Impact Explanation: High, because this leads to loss of solver funds used to fulfill the intent.

Likelihood Explanation: Medium, because this is only applicable for intents created on source chains whose domain IDs are different from their chain IDs depending on the underlying prover framework. However, this risk is documented by both Hyperlane and Metalayer.

Recommendation: Consider using a mapping from domain IDs to chain IDs provided by the underlying prover framework.

Eco: Fixed in PR 252.

Cantina Managed: Reviewed that PR 252 removes the incorrect unconditional conversion of chain to domain IDs. Instead, two new functions `_convertChainID()` and `_convertDomainID()` are added in `MessageBridgeProver` which make a hardcoded exception for `RARICHAIN` that is the only one currently for which the chain and domain IDs are different. It is noted that any future support for other such chains (with differing chain and domain IDs) will require contract changes. It is also noted that this is currently assumed for all message based provers and so any prover added in future that uses `MessageBridgeProver` will inherit this behavior.

Based on a future project update, we reviewed that 5425cbfc applies the `RARICHAIN` special-casing only to Hyperprover, not Metaprover by:

- Removing the convert `domainID<>chainID` functions and their special-casing of RARICHAIN from MessageBridgeProver.sol, which would have applied to all message-based provers.
- Moving the special-casing of `RARICHAIN` only to provers that need it, i.e. `HyperProver.sol` by converting `RARICHAIN_DOMAIN_ID` to `RARICHAIN_CHAIN_ID` while receiving proofs in `handle()` and using

`domain = _convertChainID(_sourceChainID)` in `_formatHyperlaneMessage()` while sending proofs via `prove()`.

- Casting `domain = _sourceChainID.toUint32();` without any special-casing in `MetaProver.sol` because the `RARICHAIN` disparity doesn't exist there per their update.

3.2 Medium Risk

3.2.1 Assumed instant finality may lead to loss of intent creator funds in reorg scenarios

Severity: Medium Risk

Context: MetaProver.sol#L143-L150

Summary: Assuming instant finality in `MetaProver` may lead to loss of intent creator funds in reorg scenarios, where a reorg could undo a fulfilled intent on the destination after it has been instantly proved cross-chain, in which case the solver will be rewarded on the origin without the creator's intent being fulfilled.

Finding Description: For intents using `MetaProver`, `FinalityState.INSTANT` is assumed in the call to `IMetalayerRouter(ROUTER).dispatch()` in `MetaProver.prove()`. While Caldera MetaLayer's documentation appears to be out of date (with their current Eco integration), the available documentation's Security Considerations notes the below about the risk from reorgs:

> Consider finality requirements for your use case. If your contract needs to wait for finality on the source chain before processing a message, set the _useFinalized flag to true in the dispatch function. This significantly reduces the risk of a reorg in your contract.

If `FinalityState.INSTANT` assumes near instant finality on the source chain, the MetaLayer dispatch is assumed to send the cross-chain proof message immediately (subject to block latency configurations specified in default ISMs) after the executing block without waiting for any further block confirmations.

However, if that block contained a `fulfillAndProve()` transaction and gets reorg'ed on the destination chain then the intent is not fulfilled on the destination but it's proof has already been sent by MetaLayer to the origin chain where it gets processed to allow the intent solver to claim rewards.

Impact Explanation: High, because intent solver gets rewards on the source chain while its fulfillment has been reorg'ed on the destination i.e. solver retains fulfillment funds on destination while creator loses rewards funds on the source to the solver.

Likelihood Explanation: Low (on chains like Ethereum), because while deep reorgs (>1 block) are rare, shallow reorgs (1 block) happen often, as tracked in `https://etherscan.io/blocks_forked`.

Medium (on chains like Polygon), where deep reorgs happen more often, per polygonscan.

Any `fulfillAndProve()` transaction included in such a block will lead to loss of intent creator funds if the solver is able to observe the reorg and then cancel their transaction from being included in future blocks. Malicious solvers of high-value intents may be strongly motivated to trigger such a scenario.

Recommendation: Consider enforcing MetaLayer's security guidance by using an appropriate alternative to `FinalityState.INSTANT` that waits for a certain number of block confirmations to assume finality.

Eco: We feel confident that the block latency configurations specified in default ISMs for both Hyperlane and Metalayer are sufficiently safe and best optimize for the combination of security and speed that the protocol is aiming for.

Cantina Managed: Acknowledged.

3.2.2 Incorrect check for prover calls may lead to DoS for intent fulfillment

Severity: Medium Risk

Context: Inbox.sol#L211-L220

Summary: Incorrect check to prevent prover calls may lead to DoS for intent fulfillment if any of the intent targets implement IERC165.

Finding Description: During fulfillment, we need to check if the intent is attempting to call any of the provers to send an unauthorized message where claimant addresses could be set for arbitrary intents to steal intent creator funds without actually fulfilling intents. This check is implemented in `_fulfill()` by calling the intent target with a function signature of `supportsInterface(bytes4)` and `IPROVER_INTERFACE_ID` argument.

However, this check is implemented incorrectly because it reverts after checking only that the call's `success` bool is true without also checking that the return data value is true. This incorrect checking logic will therefore revert for all IERC165 targets implementing the `supportsInterface(bytes4)` function, which includes the provers (implementing `IPROVER_INTERFACE_ID`) but also potentially valid intent targets (not implementing `IPROVER_INTERFACE_ID`).

Note: This issue was identified and reported by the Project team after the review began.

Impact Explanation: Medium, because this will prevent valid intent targets implementing the `supportsInterface(bytes4)` function from ever being fulfilled and therefore grief solvers.

Likelihood Explanation: Medium, because it is likely that intent targets may be `IERC165` compliant implementing the `supportsInterface(bytes4)` function.

Recommendation: Consider fixing the prover checking logic to check that the call's `success` bool is true and also that the return data value is true.

Eco: The initial fix in PR 242, had an issue where intents to send native tokens to EOAs would always fail. This was later mitigated in PR 257.

Cantina Managed: Reviewed that PR 242 and PR 257 fix the issue by replacing the logic with `try/catch` block.

3.3 Low Risk

3.3.1 Front-running intent fulfillment transactions result in the loss of solver-approved funds

Severity: Low Risk

Context: Inbox.sol#L158

Description: The `Inbox.sol` contract is the entry point for fulfilling intents on the destination chain. To fulfill an intent, a solver should submit the user intent-related information to either one of these three functions: `fulfill()`, `fulfillAndProve()`, and `fill()`.

These three functions, in turn, use an internal function named `_fulfill()` to complete the process, as the `Inbox.sol` contract has no visibility about the intent source, the fulfillment process is purely optimistic.

In its intent fulfillment process, the `Inbox.sol` contract allows arbitrary external calls, which could be weaponized to steal funds from solvers who have already approved tokens to the Inbox contract but haven't yet executed their fulfillment transaction.

Unlike a traditional front-running attack that merely replaces the beneficiary of a transaction, this vulnerability is more severe as it enables direct theft of approved funds through carefully crafted malicious calls.

The core issue lies in the `_fulfill()` function, which executes arbitrary calls provided in the Route structure:

```
(bool success, bytes memory result) = call.target.call{
   value: call.value
 }(call.data);
    if (!success) {
        revert IntentCallFailed(
          call.target,
          call.data,
          call.value,
          result
     );
}
```

Proof of Concept: As demonstrated in the `test_frontRunning()` function, the attack proceeds as follows:

- A legitimate solver approves tokens to the `Inbox.sol` contract (e.g., `token.approve(address(inbox), 1e18)`).
- Before the solver can execute their fulfillment transaction, a malicious actor observes this approval in the mempool.
- The attacker front-runs with a malicious transaction that calls the `fulfill()` function with crafted parameters:
  - The `route.calls` array includes a call to the token contract with `transferFrom` to move funds from the solver to the attacker.
- The attacker constructs a valid intent hash that passes the verification checks.
- The contract executes the `transferFrom` call with the solver's existing approval, effectively stealing their funds.
- When the solver's legitimate transaction attempts to execute, it may fail due to insufficient funds / approvals.
- The attack exploits explicitly the fact that the contract executes arbitrary calldata against any contract (except provers), allowing attackers to target previously approved tokens.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import {Test} from "forge-std/Test.sol";
import {Inbox} from "contracts/Inbox.sol";
import {Route, Call} from "contracts/types/Intent.sol";

import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract Token is ERC20 {
    constructor() ERC20("Token", "TKN") {
        _mint(msg.sender, 100 ether);
    }
}

contract AuditTest is Test {
    Inbox public inbox;
    Token public token;

    address solver = makeAddr("solver");
    address mal = makeAddr("maliciou");

    function setUp() external {
        inbox = new Inbox();

        vm.startPrank(solver);
        token = new Token();
        vm.stopPrank();
    }

    function test_frontRunning() public {
        vm.startPrank(solver);

        token.approve(address(inbox), 1e18);

        Route memory route;
        route.salt = keccak256("non-existent-salt");
        route.source = 1;
        route.destination = block.chainid;
        route.inbox = address(inbox);
        route.tokens = new TokenAmount[](0);

        Call memory call;
        call.target = address(token);
        call.data = abi.encodeWithSelector(ERC20.transferFrom.selector, address(solver), address(mal), 1e18);

        Call[] memory calls = new Call[](1);
        calls[0] = call;

        route.calls = calls;

        bytes32 expectedHash =  keccak256(abi.encodePacked(
```

```
            keccak256(abi.encode(route)), keccak256("non-existent-reward-hash")
        ));

        /// malicious actor front-running
        inbox.fulfill(route, keccak256("non-existent-reward-hash"), mal, expectedHash, address(100));

        /// solver's actual fulfil transaction

        vm.stopPrank();
    }
}
```

Recommendation:

1. Consider removing arbitrary external calls:
    - Follow the pattern used by message bridges like Hyperlane.
    - Deliver messages to a predetermined interface (e.g., a `handle()` function).
    - This prevents execution of unexpected calldata against arbitrary contracts.

```
// Example of safer approach with fixed interface
interface IIntentHandler {
    function handleIntent(bytes memory intentData) external returns (bool);
}

function _fulfill(...) internal {
    // Verification logic...

    // Only call a specific function on the target
    IIntentHandler(intent.target).handleIntent(intent.data);
}
```

2. Enhance Documentation and User Guidelines:
    - Document this risk for solvers.
    - Recommend fulfilling all intents in a single atomic transaction.
    - Advise using private MEV-protected RPC nodes.
    - Suggest using services like Flashbots to protect transactions.

Eco: Fixed in PR 245.

Cantina Managed: Reviewed that PR 245 adds security guidelines/warnings in the documentation that includes some of what's recommended in (2).

3.3.2 Prover address mismatch between source and destination chains can lead to DoS

Severity: Low Risk

Context: Eco7683DestinationSettler.sol#L42

Description: The `fill()` function of `Eco7683DestinationSettler` uses `intent.reward.prover`—an address from the source chain—directly as the `_localProver's parameter in the` fulfillAndProve()' function on the destination chain's Inbox contract.

This function assumes that the prover addresses on the source and destination chains will be the same. This design will fail on chains like zkSync, where address derivation methods differ from traditional EVM chains. As a result, to support such chains, all the `Inbox` contracts should be redeployed, or the function `fill()` will remain unusable for those routes.

Recommendation: Consider creating a mapping of local prover to a remote prover and reference that in the `fill()` function:

```
mapping(uint256 => address) public remoteProver;

function fill(
    bytes32 _orderId,
    bytes calldata _originData,
    bytes calldata _fillerData
) external payable {
   // ...
   fulfillAndProve(
      intent.route,
      rewardHash,
      claimant,
      _orderId,
      remoteProver[intent.route.source],
      data
   );
}
```

Eco: Fixed in PR 240.

Cantina Managed: Reviewed that PR 240 replaces the use of `intent.reward.prover` with `localProver` derived from `_fillerData` provided by the caller.

3.3.3 Missing chain-specific prover authorizations may allow spoofed proofs from unsupported chains to compromise all protocol intents

Severity: Low Risk

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

Description: `MetaProver.sol` and `HyperProver.sol` do not enforce chain-specific authorization for provers. While they validate that incoming messages originate from trusted components (e.g., a Mailbox or Router) and ensure the sender prover contract is in a global whitelist, they fail to validate whether the sender is authorized for the specific origin chain of the message.

This issue lies primarily in `_handleCrossChainMessage()` of `MessageBridgeProver.sol` abstract contract, which receives a `_sourceChainId` parameter that is entirely unused in the validation process—commented out in the signature and ignored in logic.

```
function _handleCrossChainMessage(uint256, /* _sourceChainId */ address _messageSender, bytes calldata
↪  _message)
    internal
{
    if (!isWhitelisted(_messageSender)) {
        revert UnauthorizedIncomingProof(_messageSender);
    }
    // Missing: check that _messageSender is authorized for _sourceChainId
}
```

As a result, any globally whitelisted prover address can send cross-chain messages from any chain, bypassing intended trust boundaries. This enables the below exploit scenario:

1. A prover authorized only for chain A is globally whitelisted.
2. The prover deployer's credentials (private key + salt) are compromised.
3. Attacker redeploys the same prover contract (same address via CREATE3) on chain B, which is currently unsupported, along with Inbox (an EOA controlled by attacker and used in prover deployment) and other tooling required.
4. Attacker monitors all intents meant to be fulfilled on chain A and sends spoofed proofs from chain B where they control the Inbox address. Because `MessageBridgeProver` on the intent origin chain does not validate the message's source chain, the message from chain B will be incorrectly accepted as valid.

This is especially dangerous for Eco, which uses deterministic deployment (e.g., CREATE3) to replicate contract addresses across chains. Differences in the constructor parameters (including Inbox address) will result in the same prover address.

Likelihood Explanation: Very low, because this requires the compromise of the prover deployer's credentials (private key + salt), whose likelihood depends on the operational security of the deployer key and system. Assuming a hardened isolated system and the use of an effective Multisig, this should be very low.

Impact Explanation: High, because this allows the attacker to falsely claim rewards for every intent on every chain supported by the protocol, without fulfilling any of them. Without any provision to prevent/pause the attacker from doing so, the protocol will be entirely compromised.

Proof of Concept: This test case shows that CREATE3 deploys a prover to the same address with a different constructor argument (inbox address):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import {Test, console} from "forge-std/Test.sol";
import {Inbox} from "contracts/Inbox.sol";
import {HyperProver} from "contracts/prover/HyperProver.sol";

import {ICreate3Deployer} from "contracts/tools/ICreate3Deployer.sol";

contract AuditTest is Test {
    HyperProver public prover;
    Inbox public inbox;
    ICreate3Deployer public create3deployer = ICreate3Deployer(0xC6BAd1EbAF366288dA6FB5689119eDd695a66814);

    address public claimant = makeAddr("claimant");
    address public deployer = makeAddr("deployer");

    address public malInbox = makeAddr("malicious inbox");
    address public mockInbox = makeAddr("mock inbox");

    function setUp() external {}

    function test_create() public {
        vm.createSelectFork(vm.envString("BASE_RPC"));
        vm.startPrank(deployer);

        inbox = new Inbox();
        prover = new HyperProver(0xea87ae93Fa0019a82A727BFd3EBD1cFcA8F64a1d, address(inbox), new
        ↪  address[](20), 200_000);
        console.log("Prover on base:", address(prover));

        vm.createSelectFork(vm.envString("OPTIMISM_RPC"));
        inbox = new Inbox();
        prover =
            new HyperProver(0xd4C1905BB1D26BC93DAC913e13CaCC278CdCC80D, address(malInbox), new address[](20),
            ↪  200_000);

        console.log("Prover on OP:", address(prover));
    }

    function test_create2() public {
        vm.createSelectFork(vm.envString("BASE_RPC"));
        vm.startPrank(deployer);

        inbox = new Inbox{salt: keccak256("hacked-keys")}();
        prover = new HyperProver{salt: keccak256("hacked-keys")}(
            0xea87ae93Fa0019a82A727BFd3EBD1cFcA8F64a1d, address(inbox), new address[](20), 200_000
        );
        console.log("Prover on Base:", address(prover));

        vm.createSelectFork(vm.envString("OPTIMISM_RPC"));
        inbox = new Inbox{salt: keccak256("hacked-keys")}();
        prover = new HyperProver{salt: keccak256("hacked-keys")}(
            0xd4C1905BB1D26BC93DAC913e13CaCC278CdCC80D, address(malInbox), new address[](20), 200_000
        );

        console.log("Prover on OP:", address(prover));
    }

    function test_create3() public {
        vm.createSelectFork(vm.envString("BASE_RPC"));
        vm.startPrank(deployer);
```

```
        bytes memory constructorArgs = abi.encode(
            0xea87ae93Fa0019a82A727BFd3EBD1cFcA8F64a1d,
            address(mockInbox),
            new address[](0)
        );

        bytes memory bytecode = abi.encodePacked(
            type(HyperProver).creationCode,
            constructorArgs
        );

        create3deployer.deploy(bytecode, keccak256("hacked-keys"));
        address deployedContract = create3deployer.deployedAddress(
            bytecode,
            deployer,
            keccak256("hacked-keys")
        );
        console.log("Prover on base:", address(deployedContract));

        vm.createSelectFork(vm.envString("OPTIMISM_RPC"));
        constructorArgs = abi.encode(
            0xd4C1905BB1D26BC93DAC913e13CaCC278CdCC80D,
            address(malInbox),
            new address[](0)
        );

        bytecode = abi.encodePacked(
            type(HyperProver).creationCode,
            constructorArgs
        );

        create3deployer.deploy(bytecode, keccak256("hacked-keys"));
        deployedContract = create3deployer.deployedAddress(
            bytecode,
            deployer,
            keccak256("hacked-keys")
        );
        console.log("Prover on OP:", address(deployedContract));
    }
}
```

Recommendation: Consider:

1. Implementing chain-specific validation of prover addresses to ensure incoming messages are from supported chains. This will require authorized updates to such a whitelist or redeployment any time a new chain is supported.
2. Ensure that the deployer credentials have the highest degree of operational security and system security enforced.

Eco: Fixed in PR 252.

Cantina Managed: Reviewed that PR 252 refactors several states/flows significantly to address this concern as follows:

1. `_provenIntents` stores `destinationChainID` along with the `claimant`.
2. `_handleCrossChainMessage` receives a `_destinationDomainID` which is considered instead of being ignored.
3. `_processIntentProofs()` checks `destinationChainID` of incoming proofs.
4. A malicious prover deployed on an unsupported destination chain can claim to have solved an intent by proving it once. But an honest solver, which solves the intent on the intended destination chain, can observe that its proof failed on the source chain (because of the malicious solver's previously registered proof) and that its `withdrawRewards()` attempt failed but challenged the intent proof to successfully clear the previously registered malicious claimant. Thereafter, the honest solver is expected to resubmit its proof which should succeed and allow it to withdraw its rewards.
5. A malicious prover can only be successful in doing the above once for any intent because `challengeIntentProof()` is a public function that can be called by anyone, and is called in `withdrawRewards()` (by the intent's honest solver) which clears the `claimant` and registers `intent.route.destination`, preventing `_processIntentProofs()` from processing any more

malicious proofs (from destination chains different from `intent.route.destination`) for that intent.

Given that these changes impact parts of code that were initially out-of-scope for the review, our best effort in reviewing these flows during the fix period indicates that while this appears to mitigate the issue there may be:

1. Some griefing vectors and timing attacks. For e.g., consider a scenario where a honest solver fulfills and attempts to prove. A malicious prover front-runs from an unsupported chain which will revert the honest solver's proof. Honest solver attempts to withdraw rewards which fails but instead sees `IntentProofChallenged` event (assuming they're monitoring). It has to then reprove on the destination chain to claim its rewards but all this has to happen before the intent deadline expires because otherwise the intent creator may refund the reward leading to a loss of solver funds. So, solvers have to now factor in this additional possibility before solving any intent. They may also challenge before solving an intent to preempt malicious provers.
2. Other latent issues in interactions with code out-of-scope for this review.

3.4 Informational

3.4.1 Redundant code constructs reduce readability

Severity: Informational

Context: Eco7683DestinationSettler.sol#L7, Eco7683DestinationSettler.sol#L10, Inbox.sol#L6, Inbox.sol#L40, Inbox.sol#L103-L105, IMessageBridgeProver.sol#L65, HyperProver.sol#L7, HyperProver.sol#L124-L134

Description: Multiple contracts have unused file imports, errors and variable declarations that could be removed. This will help improve code quality and readability.

Recommendation: Consider removing unused code constructs.

Eco: Fixed in PR 243.

Cantina Managed: PR 243 removes most of the redundant code constructs as recommended.

3.4.2 Missing source chain prover validation may lead to lost messages

Severity: Informational

Context: HyperProver.sol#L300, MetaProver.sol#L95

Description: Both `HyperProver.sol` and `MetaProver.sol` lack validation to ensure that the source chain prover address (encoded as bytes32) is not zero before dispatching cross-chain messages. This could potentially allow a message to be dispatched to the zero address on the destination chain if a zero address is accidentally provided in the `_data` parameter.

Recommendation: Consider adding explicit validation to check that the source chain prover address is not zero before using it as a recipient:

```
function prove(
   address _sender,
   uint256 _sourceChainId,
   bytes32[] calldata _intentHashes,
   address[] calldata _claimants,
   bytes calldata _data
) external payable override {
   // ...

   // Decode source chain prover address only once
   bytes32 sourceChainProver = abi.decode(_data, (bytes32));

   if(sourceChainProver == bytes32(0)) revert INVALID_SOURCE_PROVER();

  /// ...
}
```

Eco: It would fall upon the solver to ensure that the prover address an intent creator provides points to an actual valid prover. If it were instead a malicious prover that would just ignore hyperlane messages the solver would be in the same situation. Fixed in PR 245.

Cantina Managed: Acknowledged. Reviewed that PR 245 adds security guidelines/warnings in the documentation, which includes prover checks.

3.4.3 Stale prover documentation reduces integration due-diligence

Severity: Informational

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

Description: The documentations of Hyperlane and Metalayer appear to be stale/outdated compared to their implementations being integrated within the protocol. This reduces the due-diligence by not allowing a deeper-dive into potential integration issues. For example:

1. Hyperlane's `quoteDispatch` signature used is:

   ```
   IMailbox(MAILBOX).quoteDispatch(
       destinationDomain,
       recipientAddress,
       messageBody,
       metadata,
       hook
   );
   ```

   whereas their documentation specifies it as:

   ```
   function quoteDispatch(
       uint32 destinationDomain,
       bytes32 recipientAddress,
       bytes calldata messageBody
   ) external view returns (uint256 fee);
   ```

2. Hyperlane's `dispatch` signature used is:

   ```
   IMailbox(MAILBOX).dispatch{value: fee}(
       destinationDomain,
       recipientAddress,
       messageBody,
       metadata,
       hook
   );
   ```

   whereas their documentation specifies it as:

   ```
   function dispatch(
       uint32 destinationDomain,
       bytes32 recipientAddress,
       bytes calldata messageBody
   ) external payable returns (bytes32 messageId);
   ```

3. Metalayer's `quoteDispatch` signature used is:

   ```
   IMetalayerRouter(ROUTER).quoteDispatch(domain, recipient, message);
   ```

   whereas their documentation only specifies `quoteGasPayment()` but not `quoteDispatch()`.

4. Metalayer's `dispatch` signature used is:

   ```
   IMetalayerRouter(ROUTER).dispatch{value: fee}(
       domain,
       recipient,
       new ReadOperation[](0),
       message,
       FinalityState.INSTANT,
       gasLimit
   );
   ```

   whereas their documentation specifies it as:

```
function dispatch(
    uint32 _destinationDomain,
    address _recipientAddress,
    ReadOperation[] memory _reads, // can be empty
    bytes memory _writeCallData,
    bool _useFinalized
) external payable;
```

Recommendation: Consider harmonizing the implementation with the documentation to carefully ensure no integration issues.

Eco: Acknowledged.

Cantina Managed: Acknowledged.

3.4.4 Allowing arbitrary custom hooks to be specified by the solver/prover may be risky

Severity: Informational

Context: HyperProver.sol#L144-L155, HyperProver.sol#L160-L166, HyperProver.sol#L308-L311

Description: Solvers/provers using Hyperlane are allowed to specify a post dispatch hook. Hyperlane's hook documentation says:

> Post-dispatch hooks allow developers to configure additional origin chain behavior with message content dispatched via the Mailbox. This allows developers to integrate third party/native bridges, make additional chain commitments, or require custom fees all while maintaining a consistent single-call Mailbox interface.

Eco plans to use hooks to run their own Hyperlane relayer, the details of which are out-of-scope for this review. However, allowing arbitrary custom hooks to be specified by the solver/prover may be risky because hooks work closely with Hyperlane's Interchain Security Modules (ISMs).

Recommendation: Consider a security review of the use of Hyperlane hooks with the relayer integration.

Eco: We've kept it this way since we're still working on the custom relayer, but yeah once we get that sorted this may become something more strict that lives in storage and we just pull in. for now going to leave it though, as the solver set is very small and sophisticated.

Cantina Managed: Acknowledged.

3.4.5 Assuming that all solvers will be sophisticated to perform detailed security checks on intents may be risky

Severity: Informational

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

Description: While the protocol implements some defensive checks to ensure that solvers engage with intents in expected ways, there are flows where some security checks are missing (apparently to reduce execution overhead) and solvers are expected to be sophisticated enough to themselves perform such checks. For example:

1. Solvers are expected to check that provers used by intents are trustworthy and not malicious.
2. Solvers are expected to atomically approve tokens and fulfill intents to prevent front-running.
3. Solvers are expected to check reward deadlines and only fulfill valid intents where they have the time to prove and claim rewards.

Assuming that all solvers will be sophisticated enough to perform detailed security checks on all intents may be risky. Malicious intent creators will be motivated to trick solvers from fulfilling intents where they are prevented from claiming their subsequent rewards because of failing to perform sufficient due-diligence on the intents. Such incidents may negatively impact protocol reputation and solver participation.

Recommendation:

1. Consider adding as many defensive checks as possible to prevent solvers (and intent creators) from getting exploited.
2. Consider adding guidelines on security considerations and best practices for solvers to facilitate such checks.

Eco: The defensive checks being put in will only solve for common pitfalls, but will not prevent malicious intent creators from setting traps (malicious provers, very short-term intents, bad reward tokens). Given that the system demands a high level of vigilance no matter what, we figured we can trust solvers to avoid simple mistakes and in so doing save them some gas. Fixed in PR 245.

Cantina Managed: Acknowledged. Reviewed that PR 245 adds security guidelines/warnings in the documentation.

3.4.6 Return bomb during fulfillment may grief solvers

Severity: Informational

Context: Inbox.sol#L221

Description: An attacker can create a malicious contract that returns a massive data payload when called via the `_fulfill()` function. The contract will attempt to allocate since the function stores this returned data in the results array without size checks.

This can cause the solver to pay excessively large gas values for the fulfill call. The transaction can sometimes revert, leading to gas wastage for the solvers.

Recommendation: Consider using libraries like `ExcessivelySafeCall` to limit the return value to be copied to avoid excessive memory allocation, thereby mitigating the return bomb issue.

Eco: Malicious contracts are something the solver will no doubt have to look out for before fulfilling an intent. Given that this is just one of a myriad of different attacks and we will anyway be creating documentation to help protect solvers, I don't think it's worth spending the gas to stop this one specific attack. Fixed in PR 245.

Cantina Managed: Acknowledged. Reviewed that PR 245 adds security guidelines/warnings in the documentation, which includes malicious intents.

3.4.7 Incorrect enforcement of minimum gas limit override in `MetaProver` may lead to message delays/failures

Severity: Informational

Context: MetaProver.sol#L125

Description: In the `MetaProver.sol`, when a custom gas limit is provided in the `_data` parameter, the contract fails to properly validate that this value is sufficient to guarantee successful message delivery.

```
// For Metalayer, we expect data to include sourceChainProver(32 bytes)
// If data is long enough, the gas limit is packed at position 32-64
if (_data.length >= 64) {
    uint256 customGasLimit = uint256(bytes32(_data[32:64]));
    if (customGasLimit > 0) {
        gasLimit = customGasLimit;
    }
}
```

The contract allows any positive gas limit value to be used without verifying that it meets the minimum requirements for successful message delivery. This could lead to message dispatch with insufficient gas, causing transaction failures on the destination chain, and a temporary delay in message proving.

Recommendation: Consider implementing proper validation for custom gas limits:

```
if (_data.length >= 64) {
    uint256 customGasLimit = uint256(bytes32(_data[32:64]));
    if (customGasLimit > DEFAULT_GAS_LIMIT) {
        gasLimit = customGasLimit;
    }
}
```

Eco: Fixed in PR 244 and PR 252.

Cantina Managed: Reviewed that PR 244 and PR 252 fix the issue as recommended.