# Optimism Interop Security Review


**Auditors**

Zach Obront, Lead Security Researcher

Phaze, Lead Security Researcher

RustyRabbit, Security Researcher

Rvierdiiev, Associate Security Researcher

**Report prepared by:** Lucas Goiriz



March 19, 2025

# Contents

# 1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

# 2 Introduction

Optimism is a fast, stable, and scalable L2 blockchain built by Ethereum developers, for Ethereum developers. Built as a minimal extension to existing Ethereum software, Optimism's EVM-equivalent architecture scales your Ethereum apps without surprises. If it works on Ethereum, it works on Optimism at a fraction of the cost.

*Disclaimer*: This security review does not guarantee against a hack. It is a snapshot in time of Optimism Interop according to the specific commit. Any modifications to the code will require a new security review.

# 3 Risk classification

| Severity level | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| **Likelihood: high** | Critical | High | Medium |
| **Likelihood: medium** | High | Medium | Low |
| **Likelihood: low** | Medium | Low | Low |

## 3.1 Impact

- 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.

## 3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

## 3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

# 4 Executive Summary

Over the course of 18 days in total, OP Labs engaged with Spearbit to review the optimism-interop protocol. In this period of time a total of **20** issues were found.

**Summary**

| | |
|---|---|
| **Project Name** | OP Labs |
| **Repository** | optimism-interop |
| **Commit** | 6c80f23a |
| **Type of Project** | DeFi, AMM |
| **Audit Timeline** | Feb 11th to Mar 1st |

**Issues Found**

| Severity | Count | Fixed | Acknowledged |
|---|---|---|---|
| Critical Risk | 0 | 0 | 0 |
| High Risk | 2 | 0 | 2 |
| Medium Risk | 3 | 0 | 3 |
| Low Risk | 4 | 0 | 4 |
| Gas Optimizations | 3 | 0 | 3 |
| Informational | 8 | 2 | 6 |
| **Total** | **20** | **2** | **18** |

# 5 Findings

## 5.1 High Risk

### 5.1.1 Chain operator can DOS entire cluster during upgrade block, potentially stealing ETH from other chains

**Severity:** High Risk

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

**Description:** In the Interop Network Upgrade (as well as other OP hard forks that include L2 contract changes), the upgrade block is organized as follows:

1) L1 Attributes Transaction calling `setL1BlockValuesEcotone`.
2) User deposits from L1.
3) Network upgrade transactions.

It is required that the block has sufficient gas for all of these transactions, because they are all deposit transactions and therefore must be included in the block. If this is not the case, we will get an `ErrGasLimitReached`. This will cause a crash of all `op-node` and `op-program` instances, halting the chain and making fault proofs impossible to prove.

At a minimum, this will DoS the entire cluster. Depending how interop fault proofs are implemented, it also may create the opportunity for the malicious chain operator to propose a malicious root that cannot be disputed. In this case, they could create a root that include a withdrawal of all ETH in the `SharedLockbox`, stealing funds from the other chains.

Is it possible for a single chain operator to create this out of gas situation? We can see in `SystemConfig.sol`, the `setGasLimit()` function is callable by the chain operator. It requires that they set the gas limit to at least the minimum:

```
function minimumGasLimit() public view returns (uint64) {
    return uint64(_resourceConfig.maxResourceLimit) + uint64(_resourceConfig.systemTxMaxGas);
}
```

However, as we can see, this minimum only takes into account the L1 attributes transaction (`systemTxMaxGas`) and user deposits (`maxResourceLimit`). It is therefore possible to reduce the gas limit down to the minimum, use all the possible gas for user deposits, and force the network upgrade transaction to run out of gas.

**Proof of Concept:**

- Let's imagine a chain has 1mm allocated for the system transaction and 20mm for user deposits.
- The total of all the upgrade transactions is 5mm gas.
- Some time before the upgrade block, the chain operator lowers the gas limit to the minimum (21mm).
- At the upgrade block, they submit a 20mm gas transaction through the bridge.
- As the upgrade block executes, it will use some portion of the 1mm for the system transaction, all 20mm for the user deposits, and not have sufficient gas left in the block for the network upgrade transactions.
- The result is that the chain halts, with op-node and op-program unable to perform derivation.

**Recommendation:** Ensure an additional buffer is added between the L2 gas limit and `systemTxMaxGas + maxResourceLimit`. Ensure all network upgrade transactions use an amount of gas that is less than this buffer.

**OP Labs:** This is being fixed in an earlier hardfork (Ishtmus) in commit 8966189c.

**Cantina Managed:** Will be fixed by changes outside of the scope.

### 5.1.2 Malicious tokens can be deployed at deterministic addresses on other chains to steal funds

**Severity:** High Risk

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

**Description:** When sending an ERC20 across the `SuperchainTokenBridge`, we assume that the token has the same address on both chains. In `sendERC20()`, we encode the sent token address in the relay call:

```
bytes memory message = abi.encodeCall(this.relayERC20, (_token, msg.sender, _to, _amount));
```

In `relayERC20()`, we call `crosschainMint()` on this address:

```
ISuperchainERC20(_token).crosschainMint(_to, _amount);
```

This means that any two SuperchainERC20 compatible tokens deployed on chains in the same cluster can be exchanged freely with one another. Currently, the plan is for these tokens to be deployed using a generic `CREATE2` factory. Given the existence of the same factory on both chains, the inputs into the deterministic generation of the token's address are the salt and the initialization code (which includes constructor arguments).

This means that any Superchain token implementation that either (a) uses a fixed address instead of constructor arguments to allocate initial tokens or ownership of the contract or (b) uses an `initialize()` function after the constructor for the same purposes, will be deployed to identical addresses with different outcomes.

Note that this is near impossible to avoid because, each time a new chain is added to the interop set, the attack becomes possible. An astute attacker could predeploy token contracts on potential interop chains so that, upon interop being enabled, they would get unlimited minting rights on the legitimate tokens on other chains.

**Proof of Concept:** The following test demonstrates the example where funds are sent in an `initialize()` function. Note that the same attack is possible if ownership is allocated this way, or if these values are set to hardcoded values (such as Gnosis Safes) that may be claimable on the other network.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import { Test, console } from "forge-std/Test.sol";
import { SuperchainERC20 } from "@optimism/L2/SuperchainERC20.sol";

interface Create2Factory {
    function deploy(uint256 value, bytes32 salt, bytes memory code) external;
    function computeAddress(bytes32 salt, bytes32 codeHash) external view returns(address);
}

contract SuperchainUSDC is SuperchainERC20 {
    function initialize() external {
        require(totalSupply() == 0, "already initialized");
        _mint(msg.sender, 100_000_000e18);
    }

    function name() public pure override returns (string memory) {
        return "Superchain USDC";
    }

    function symbol() public pure override returns (string memory) {
        return "USDC";
    }
}

contract Create2Test is Test {
    Create2Factory CREATE2 = Create2Factory(0x13b0D85CcB8bf860b6b79AF3029fCA081AE9beF2);

    function testCreate2Address() external {
        address honest = address(uint160(uint256(keccak256("honest user"))));
```

```
        address malicious = address(uint160(uint256(keccak256("malicious user"))));

        vm.createSelectFork("https://optimism-mainnet.infura.io/v3/fb419f740b7e401bad5bec77d0d285a5");

        // On the original chain, a token is deployed with CREATE2 factory.
        uint before = vm.snapshot();
        bytes memory initCode = type(SuperchainUSDC).creationCode;
        vm.startPrank(honest);
        CREATE2.deploy(0, bytes32(0), initCode);

        SuperchainUSDC honestToken = SuperchainUSDC(CREATE2.computeAddress(bytes32(0),
        ↪  keccak256(initCode)));
        honestToken.initialize();
        assertEq(honestToken.balanceOf(honest), 100_000_000e18);

        // On the new chain, a malicious user deploys a token with the same code and claim the assets.
        vm.revertTo(before);
        vm.startPrank(malicious);

        CREATE2.deploy(0, bytes32(0), initCode);
        SuperchainUSDC maliciousToken
            = SuperchainUSDC(CREATE2.computeAddress(bytes32(0), keccak256(initCode)));
        maliciousToken.initialize();
        assertEq(maliciousToken.balanceOf(malicious), 100_000_000e18);
    }
}
```

**Recommendation:** `SuperchainToken` developers must be made aware of the fact that all important values must be set in the token's constructor to ensure only an identical deployment with identical properties can be deployed to the same address.

**OP Labs:** Acknowledged. From the documentation:

> To ensure security, you must either design the deployer to allow only a specific trusted ERC-20 contract, such as SuperchainERC20, to be deployed through it, or call CREATE2 to deploy the contract directly from an EOA you control. This precaution is critical because if an unauthorized ERC-20 contract is deployed at the same address on any Superchain network, it could allow malicious actors to mint unlimited tokens and bridge them to the network where the original ERC-20 contract resides.

**Cantina Managed:** Acknowledged.

## 5.2 Medium Risk

### 5.2.1 Withdrawals could be bricked if chain enabled interop before migrating to the `SharedLockbox`

**Severity:** Medium Risk

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

**Description:** Upgrading an L2 chain to enable interop (and join the dependency set of its peers) is a distinct process from updating the L1 bridge (and moving ETH to the `SharedLockbox`). The intention is for these two processes to be decoupled. After interop is enabled on an L2, the possibility exists for more ETH to be withdrawn from the chain than exists in the L1 OptimismPortal. This occurs because ETH can be bridged from other chains. This is the problem that the SharedLockbox solves.

However, if there is a time period where interop is enabled and funds have not been migrated to the SharedLockbox, we can end up with withdrawals that cannot be fulfilled by the balance of the Portal. Ideally, in this situation, these withdrawals would revert. However, the result is actually that withdrawals will be finalized and fail to execute, bricking the user's funds.

**Proof of Concept:** If a withdrawal was performed in the situation described above:

- Anyone can call `finalizeWithdrawalTransaction()` on behalf of any other withdrawal.

- The call to `_unlockETH()` will be a no op (if the portal has not been upgraded), or will return early (if it has been upgraded but not yet migrated).
- There will be less funds in the Portal than required for the `tx.value`.
- The `SafeCall.callWithMinGas()` call will fail, but the transaction will not revert.
- The withdrawal will not be replayable, losing the user their funds.

To demonstrate that a `callWithMinGas()` with insufficient funds will return `success = false` rather than revert, you can use this standalone test:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

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

contract InsufficientTest is Test {
    function testInsufficient() public {
        address user = makeAddr("user");
        vm.deal(user, 1);

        vm.prank(user);
        (bool success,) = address(1).call{value: 2}("");
        console.log(success);
    }
}
```

**Recommendation:** Add a check to ensure that there is sufficient balance before calling `callWithMinGas()`, and revert if this isn't the case to maintain replayability:

```
  // This function unlocks ETH from the SharedLockbox when using the OptimismPortalInterop contract.
  // If the interop version is not used, this function is a no-ops.
  _unlockETH(_tx);

+ if (_tx.value > address(this).balance) revert NotEnoughBalance

  // Trigger the call to the target contract. We use a custom low level method
  // SafeCall.callWithMinGas to ensure two key properties
  //   1. Target contracts cannot force this call to run out of gas by returning a very large
  //      amount of data (and this is OK because we don't care about the returndata here).
  //   2. The amount of gas provided to the execution context of the target is at least the
  //      gas limit specified by the user. If there is not enough gas in the current context
  //      to accomplish this, `callWithMinGas` will revert.
  bool success = SafeCall.callWithMinGas(_tx.target, _tx.gasLimit, _tx.value, _tx.data);
```

**OP Labs:** Acknowledged. `_unlockETH` is no longer a no-op and will be part of the `OptimismPortal` due to lockbox redesign, on PR 14588.

**Cantina Managed:** Acknowledged.

### 5.2.2 Transaction prevention through invalid message validation

**Severity:** Medium Risk

**Context:** CrossL2Inbox.sol#L49

**Description:** Any contract with flow control privileges can invoke the `validateMessage` function using an invalid message hash. Since the Supervisor cannot includes such messages on the source chain, it marks these transactions as invalid and excludes them from block inclusion. This allows any contract that receives control flow to block a transaction from occurring. This is typically not the case. If there is a low level call (or a try/catch block) with a specified amount of gas, it is safe to pass control flow to an unsafe contract and be sure they cannot cause the function to revert.

This can make contracts that are safe on other chains because unsafe on Superchain chains, breaking EVM equivalence. As an example, this can be problematic during liquidation or cancellation events, where systems sometimes notify users through a `try/catch` callback mechanism, allowing them to handle this event without the ability to cause transaction reversion. However, this also enables the user to deliberately prevent liquidations by exploiting the message validation process.

**Proof of Concept:**

1. A liquidator initiates a liquidation against a user's loan position.
2. The lending contract notifies the user's designated contract through a `try/catch` mechanism.
3. The user's contract responds by calling `validateMessage` with either:
    - A non-existent message hash.
    - An identifier outside the permitted dependency set.
4. The Supervisor flags the transaction as invalid, preventing its inclusion in the block, and permanently avoiding liquidation.

**Recommendation:** Addressing this issue requires careful consideration. While providing message validation suspension functionality could provide a solution for contracts that need it, this approach must be carefully designed to prevent abuse. It should also ensure that the suspension mechanism cannot be exploited to bypass validation for genuinely invalid transactions.

**OP Labs:** This will be addressed in a redesign of `CrossL2Inbox` prior to release. Acknowledged.

**Cantina Managed:** Acknowledged.

#### 5.2.3 Native token can be flash minted, breaking protocol expectations

**Severity:** Medium Risk

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

**Description:** Protocols have been designed with the assumption that there is a fixed amount of ETH in existence. For example, ETH balances are often stored as `uint96` on the knowledge that it will be impossible to overflow. The current interop contracts break this invariant by allowing users to "flash mint" ETH by calling `relayERC20()` before `sendERC20()` on the `SuperchainWETH.sol` contract. If they perform this action on two separate chains in the same block, this cycle will net out to being accepted, but will allow the caller to temporarily hold up to `type(uint248).max` of ETH mid execution.

While flash minting is a common practice with some ERC20s, it is assumed that for ETH, there is an upper bound to flash loans of the total supply in existence.

Because `type(uint248).max` of ETH will be placed in the `ETHLiquidity` contract, this assumption is no longer true, and users temporarily hold far more than the total amount of ETH in existence. This can result in important app layer assumptions being broken, breaking composability with L1 contracts.

**Proof of Concept:** We can imagine the following scenario:

- A user uses the L2ToL2CrossDomainMessenger to call `relayETH()` on the `SuperchainWETH` contract on Chain A. This gives them access to the requested WETH, but requires a sending transaction from Chain B to be valid.
- They perform any actions they would like with this ETH and then call `sendETH()` to send the funds back to Chain B, all in the same transaction.
- Meanwhile, on Chain B, they perform the same two actions (relay and then send).
- In both cases, ETH was flash minted by calling relay before send, but was valid due to the timestamp invariant.

**Recommendation:** TBD.

**OP Labs:** Acknowledged. Cycles are disallowed by the protocol. It may happen temporarily during execution, but should never happen if the sequencer checks the messages.

**Cantina Managed:** Acknowledged.

## 5.3 Low Risk

### 5.3.1 `SendETH()` allows sending ETH to invalid chain IDs

**Severity:** Low Risk

**Context:** SuperchainWETH.sol#L106-L123

**Description:** The `sendETH()` function in SuperchainWETH allows users to specify any chain ID as the destination without validation. If a user provides an invalid chain ID, either by mistake or intentionally, the ETH amount will effectively be burned as there is no way to relay it to a non-existent destination.

**Recommendation:** Consider adding a precompile that maintains a whitelist of valid destination chain IDs in the superchain network, and use it to validate destinations before allowing transfers:

```
  function sendETH(address _to, uint256 _chainId) external payable returns (bytes32 msgHash_) {
      if (_to == address(0)) revert ZeroAddress();
+     if (!IChainValidation(CHAIN_VALIDATION_PRECOMPILE).isInDependencySet(_chainId)) {
+         revert InvalidDestinationChain();
+     }

      // NOTE: 'burn' will soon change to 'deposit'.
      IETHLiquidity(Predeploys.ETH_LIQUIDITY).burn{ value: msg.value }();

      msgHash_ = IL2ToL2CrossDomainMessenger(Predeploys.L2_TO_L2_CROSS_DOMAIN_MESSENGER).sendMessage({
          _destination: _chainId,
          _target: address(this),
          _message: abi.encodeCall(this.relayETH, (msg.sender, _to, msg.value))
      });

      emit SendETH(msg.sender, _to, msg.value, _chainId);
  }
```

**OP Labs:** Acknowledged. There won't be an on-chain dependency set in this interop version, so this is impossible to fix.

**Cantina Managed:** Acknowledged.

### 5.3.2 Invalid withdrawal targets could be prevented at initiation

**Severity:** Low Risk

**Context:** OptimismPortalInterop.sol#L105-L108

**Description:** The `OptimismPortalInterop` contract validates that withdrawal transactions cannot target the SharedLockbox via `_validateWithdrawal()`. However, this check happens after the withdrawal has already been initiated on L2. A more efficient approach would be to prevent these invalid withdrawals at initiation time in the L2ToL1MessagePasser.

**Recommendation:** Add target validation to the `initiateWithdrawal()` function in L2ToL1MessagePasser:

```
  function initiateWithdrawal(address _target, uint256 _gasLimit, bytes memory _data) public payable {
+     // Check that target is not the SharedLockbox
+     // Note: The exact SharedLockbox address would need to be configured per chain
+     if (_target == SHARED_LOCKBOX_ADDRESS) revert InvalidTarget();
+     // Also prevent targeting the OptimismPortal itself
+     if (_target == OPTIMISM_PORTAL_ADDRESS) revert InvalidTarget();

      bytes32 withdrawalHash = Hashing.hashWithdrawal(
          Types.WithdrawalTransaction({
              nonce: messageNonce(),
              sender: msg.sender,
              target: _target,
              value: msg.value,
              gasLimit: _gasLimit,
              data: _data
          })
      );

      sentMessages[withdrawalHash] = true;
      emit MessagePassed(messageNonce(), msg.sender, _target, msg.value, _gasLimit, _data,
      ↪  withdrawalHash);

      unchecked {
          ++msgNonce;
      }
  }
```

Note that implementing this on L2 requires some way to determine the correct target addresses since they will be specific to each chain. This could be handled through configuration or predeploys.

**OP Labs:** Acknowledged. We don't have the portal and lockbox addresses on the L2 side, so for now we will prevent them on the portal bad target check.

**Cantina Managed:** Acknowledged.

#### 5.3.3 Chain ID parameter can be derived from permissioned game

**Severity:** Low Risk

**Context:** SuperchainConfigInterop.sol#L115-L135

**Description:** In the `addDependency()` function of SuperchainConfigInterop, the `_chainId` parameter is provided as input but could be derived directly from the permissioned game associated with the system config. This would prevent potential mismatches between the provided chain ID and the actual chain ID of the system being added.

**Recommendation:** Modify the function to derive the chain ID from the permissioned game instead of taking it as a parameter:

```
- function addDependency(uint256 _chainId, address _systemConfig) external {
+ function addDependency(address _systemConfig) external {
      if (paused()) revert SuperchainPaused();
      if (msg.sender != clusterManager()) revert Unauthorized();

      SuperchainConfigDependencies storage dependenciesStorage = _dependenciesStorage();

      if (dependenciesStorage.dependencySet.length() == type(uint8).max) revert DependencySetTooLarge();

+     // Get chain ID from permissioned game
+     IDisputeGameFactory disputeGameFactory =
↪   IDisputeGameFactory(ISystemConfig(_systemConfig).disputeGameFactory());
+     IFaultDisputeGame fdg = IFaultDisputeGame(
+         address(disputeGameFactory.gameImpls(GameTypes.PERMISSIONED_CANNON))
+     );
+     uint256 chainId = fdg.l2ChainId();

-     if (!dependenciesStorage.dependencySet.add(_chainId)) revert DependencyAlreadyAdded();
+     if (!dependenciesStorage.dependencySet.add(chainId)) revert DependencyAlreadyAdded();

      address portal = ISystemConfig(_systemConfig).optimismPortal();
      _joinSharedLockbox(portal);

-     emit DependencyAdded(_chainId, _systemConfig, portal);
+     emit DependencyAdded(chainId, _systemConfig, portal);
  }
```

This ensures that the chain ID always matches the one in the permissioned game and removes the possibility of providing an incorrect chain ID.

**OP Labs:** Acknowledged. `SuperchainConfig` changes are being deprecated.

**Cantina Managed:** Acknowledged.

#### 5.3.4 `IsDeposit()` restriction can be bypassed

**Severity:** Low Risk

**Context:** L1BlockInterop.sol#L29

**Description:** The `isDeposit()` view function has a restriction that causes it to revert when called by any contract other than `CrossL2Inbox`. However, this restriction can be circumvented using the following approach:

1. Create a wrapper contract that calls `validateMessage` with an invalid message using a `try/catch` block.
2. The wrapper contract can then:
   - If the call reverts with the `NoExecutingDeposits`, bubble up the error.
   - If the call does not revert, revert with its own custom error message (thereby avoiding the supervisor marking the transaction as invalid due to the invalid message).

This technique allows the wrapper contract to effectively signal whether a transaction is a deposit or not. Subsequently, any contract can implement try/catch logic to process deposit and non-deposit transactions differently based on these revert messages.

**Recommendation:** No solution seems viable within the normal context of the EVM.

**OP Labs:** Acknowledged. This will be addressed in a redesign of `CrossL2Inbox` prior to release.

**Cantina Managed:** Acknowledged.

## 5.4 Gas Optimization

### 5.4.1 Redundant pause checks

**Severity:** Gas Optimization

**Context:** OptimismPortal2.sol#L398-L400, SharedLockbox.sol#L74

**Description:** Pause check inside `SharedLockbox.unlockETH()` function is redundant as `OptimismPortal2.finalizeWithdrawalTransactionExternalProof()` does same check earlier in the call.

Pause check with `whenNotPaused` modifier inside `OptimismPortal2.finalizeWithdrawalTransaction(Types.WithdrawalTransaction memory _tx)` function is redundant as it is later checked with overloaded version of the function.

**Recommendation:** Consider removing extra checks to save gas.

**OP Labs:** Acknowledged. For security measures we will leave this check on the lockbox.

**Cantina Managed:** Acknowledged.

### 5.4.2 Configurable parameters that can only be changed via upgrades can be immutable

**Severity:** Gas Optimization

**Context:** SuperchainConfig.sol#L104-L110

**Description:** The `guardian` parameter in the SuperchainConfig contract and the `clusterManager` parameter in the SuperchainConfigInterop contract are only configurable during initialization and require a contract upgrade to change. This effectively makes them immutable, but it's currently stored in storage which requires more expensive storage reads compared to immutable variables.

**Recommendation:** Convert the parameter to be `immutable` and set it in the constructor of the implementation contract to better reflect its behavior and save gas:

```
  contract SuperchainConfig is Initializable, ISemver {
-     bytes32 public constant GUARDIAN_SLOT = bytes32(uint256(keccak256("superchainConfig.guardian")) -
↪  1);
+     address public immutable guardian;

      constructor(address _guardian) {
+         guardian = _guardian;
          _disableInitializers();
      }
  }
```

**OP Labs:** Acknowledged. `SuperchainConfig` changes are being deprecated.

**Cantina Managed:** Acknowledged.

### 5.4.3 Gas can be saved on all withdrawals by returning early if `tx.value == 0`

**Severity:** Gas Optimization

**Context:** OptimismPortalInterop.sol#L120-L121

**Description:** After the OptimismPortal has been upgraded to `OptimismPortalInterop`, all withdrawals call `_unlockETH()`, which pulls the ETH from the `SharedLockbox`:

```
function _unlockETH(Types.WithdrawalTransaction memory _tx) internal virtual override {
    OptimismPortalStorage storage s = _storage();

    // If ETH liquidity has not been migrated to the SharedLockbox yet, maintain legacy behavior
    // where ETH accumulates in the portal contract itself rather than being managed by the lockbox
    if (!s.migrated) return;

    // Skip calling the lockbox if the withdrawal value is 0 since there is no ETH to unlock
    if (_tx.value == 0) return;

    ISharedLockbox(s.sharedLockbox).unlockETH(_tx.value);
}
```

This call returns early if (a) it hasn't yet been migrated or (b) `tx.value == 0`. Since `tx.value == 0` will be the far more likely occurrence (since the migration is a one way transaction), and since `tx.value` doesn't require reading from storage, it would save gas to move this check to the top of the function to avoid the storage read in this case.

**Recommendation:**

```
  function _unlockETH(Types.WithdrawalTransaction memory _tx) internal virtual override {
+     // Skip calling the lockbox if the withdrawal value is 0 since there is no ETH to unlock
+     if (_tx.value == 0) return;

      OptimismPortalStorage storage s = _storage();

      // If ETH liquidity has not been migrated to the SharedLockbox yet, maintain legacy behavior
      // where ETH accumulates in the portal contract itself rather than being managed by the lockbox
      if (!s.migrated) return;

-     // Skip calling the lockbox if the withdrawal value is 0 since there is no ETH to unlock
-     if (_tx.value == 0) return;

      ISharedLockbox(s.sharedLockbox).unlockETH(_tx.value);
  }
```

**OP Labs:** Acknowledged. This changed on the current design, where we check if the value is greater than zero prior to call `unlockETH`.

**Cantina Managed:** Acknowledged.

## 5.5 Informational

### 5.5.1 Redundant event parameter for restricted caller address

**Severity:** Informational

**Context:** ETHLiquidity.sol#L31-L42

**Description:** In the ETHLiquidity contract, the `LiquidityBurned` and `LiquidityMinted` events include an indexed `caller` parameter that is always set to `Predeploys.SUPERCHAIN_WETH` since the functions are restricted to this single caller. This makes the event parameter redundant in the current implementation, though it could become useful if the contract is upgraded to support multiple callers in the future.

**Recommendation:** Consider either:

1. Remove the `caller` parameter if it's unlikely that additional callers will be supported:

```
- event LiquidityBurned(address indexed caller, uint256 value);
+ event LiquidityBurned(uint256 value);

- event LiquidityMinted(address indexed caller, uint256 value);
+ event LiquidityMinted(uint256 value);
```

2. Add a `from` parameter to track the original source of the liquidity if that information would be valuable:

```
- event LiquidityBurned(address indexed caller, uint256 value);
+ event LiquidityBurned(address indexed caller, address indexed from, uint256 value);

- event LiquidityMinted(address indexed caller, uint256 value);
+ event LiquidityMinted(address indexed caller, address indexed from, uint256 value);
```

Given that the contract could be upgraded in the future to support additional callers, keeping the current event parameters is also a reasonable choice for forward compatibility.

**OP Labs:** Acknowledged. Leaving it just in case in the future any other caller can interact with `ETHLiquidity`.

**Cantina Managed:** Acknowledged.

#### 5.5.2 Message revert reason not bubbled up on relay failure

**Severity:** Informational

**Context:** L2ToL2CrossDomainMessenger.sol#L197-L199

**Description:** The current implementation of `relayMessage()` does not properly forward the revert reason when the target call fails. By not preserving the actual revert reason, debugging becomes more difficult as the specific cause of failure is lost.

**Recommendation:** Use inline assembly to properly bubble up the revert reason from the failed call:

```
  function relayMessage(
        Identifier calldata _id,
        bytes calldata _sentMessage
  )
        external
        payable
        nonReentrant
        returns (bytes memory returnData_)
    {
        // ... previous code ...

        bool success;
        (success, returnData_) = target.call{ value: msg.value }(message);

        if (!success) {
-           revert TargetCallFailed();
+           assembly {
+               revert(add(32, returnData_), mload(returnData_))
+           }
        }

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

This preserves the original revert reason, making error investigation and debugging much easier.

**OP Labs:** Fixed in commit 44007718.

**Cantina Managed:** The fix now bubbles up the revert reason as recommended.

### 5.5.3 ETH liquidity cycling could lead to imbalanced ETH liquidity pools

**Severity:** Informational

**Context:** SuperchainWETH.sol#L106-L144

**Summary:** The ETH liquidity system can potentially be manipulated by cycling ETH through the L2ToL2CrossDomainMessenger and L2toL1MessagePasser, allowing ETH to be "*shuffled*" between chains. While this requires significant amounts to cause practical issues, an accounting imbalance could eventually lead to functionality loss in ETHLiquidity due to arithmetic overflow/underflow.

**Description:** The ETH liquidity system could be manipulated through a sequence of cross-chain messages and withdrawals to create an imbalance between different chain's ETH liquidity pools. An attacker could cycle ETH through the following path:

1. Send ETH from Chain A to Chain B using `L2ToL2CrossDomainMessenger`.
    - Chain A: Alice -1 ETH, ETHLiquidity +1 ETH.
    - Chain B: ETHLiquidity -1 ETH, Alice +1 ETH.
2. Withdraw ETH from Chain B to L1.
    - Chain B: Alice -1 ETH, SharedLockbox -1 ETH.
3. Deposit ETH back to Chain A.
    - Chain A: Alice +1 ETH, SharedLockbox +1 ETH.

Net result:

- Chain A: ETHLiquidity +1 ETH.
- Chain B: ETHLiquidity -1 ETH.

This cycle would result in Chain A's ETHLiquidity contract having more ETH than intended and Chain B having less. While this would require extremely large amounts to actually cause overflows/underflows, if such an imbalance occurred it could disrupt ETHLiquidity functionality.

**Impact Explanation:** The impact of this vulnerability is low at present due to the very large amounts of ETH required to create meaningful imbalances. However, if successfully exploited, it could eventually make ETHLiquidity unusable on affected chains, disrupting cross-chain ETH transfers.

**Likelihood Explanation:** The likelihood is very low due to:

1. The extreme amounts of ETH required.
2. The high transaction costs involved in moving ETH between chains repeatedly.
3. The time required to build up a significant imbalance.

**Recommendation:** Implement monitoring for ETH balances across chains and ETHLiquidity contracts to detect unusual patterns of cross-chain transfers. Further, develop an incident response plan that includes:

- Thresholds for concerning levels of imbalance.
- Procedures for investigating suspicious transfer patterns.
- Actions to take if significant imbalances are detected.
- Methods to correct imbalances if they occur.

**OP Labs:** Acknowledged. We should run the numbers but it seems that the cost won't be feasible.

**Cantina Managed:** Acknowledged. This scenario is entirely theoretical and should not occur in practice. Here are some rough calculations:

```
2^248 = 452312848583266388373324160190187140051835877600158453279131187530910662656
N = (2^248) / 100M = 45231284858326638837332416019018714005183587760015845327913118753091066265 6

Initial capital required: 100,000,000 ETH (reused in each cycle)
Gas cost per cycle: 0.011 ETH
Total gas cost for 45231284858326638837332416019018714005183587760015845327913118753 09 cycles:
↪  4.97544133441593e+64 ETH

Time required (1000 tx/day):
Days: 4.523128485832664e+63
Years: 1.2392132837897711e+61

Age of the universe in years: 1.38e+10
Required time compared to age of universe: 8.979806404273704e+50 times the age of the universe
```

#### 5.5.4 Misleading function name for state-changing operation

**Severity:** Informational

**Context:** CrossL2Inbox.sol#L43-L54

**Description:** The `validateMessage()` function in CrossL2Inbox does more than just validation - it also emits an event, which is a state change. The current name implies a pure validation function that only reads state, which could be misleading for developers integrating with the contract.

**Recommendation:** Consider renaming the function to better reflect that it performs a state change by emitting an event:

```
- function validateMessage(Identifier calldata _id, bytes32 _msgHash) external {
+ function processMessage(Identifier calldata _id, bytes32 _msgHash) external {
      if (IL1BlockInterop(Predeploys.L1_BLOCK_ATTRIBUTES).isDeposit()) revert NoExecutingDeposits();
      emit ExecutingMessage(_msgHash, _id);
  }
```

Alternative names could be:

- `executeMessage()`.
- `processMessage()`.

**OP Labs:** Acknowledged. Will not fix. This will be addressed in a redesign of `CrossL2Inbox` prior to release.

**Cantina Managed:** Acknowledged.

#### 5.5.5 Optional `permit2` infinite allowance support could improve UX

**Severity:** Informational

**Context:** SuperchainERC20.sol#L19

**Description:** The SuperchainERC20 contract uses Solady's ERC20 implementation (v0.0.245) which includes support for `permit2` infinite allowances but has it disabled by default. In newer Solady versions, this feature is enabled by default. Additionally, `SuperchainWETH` currently does not include `permit2` infinite allowance support.

The `permit2` infinite allowance feature can improve user experience by reducing the number of required approval transactions. This feature has been deemed safe enough to be included by default in newer versions of Solady.

**Recommendation:** Consider enabling `permit2` infinite allowances if improved UX is desired:

- For `SuperchainERC20`:

```
function _givePermit2InfiniteAllowance() internal view virtual override returns (bool) {
    return true;
}
```

- For `SuperchainWETH`, similar functionality could be added if consistent behavior is desired across all superchain tokens:

```
function allowance(address owner, address spender) public view override returns (uint256) {
    if (spender == Preinstalls.Permit2) return type(uint256).max;
    return super.allowance(owner, spender);
}
```

**OP Labs:** Acknowledged. We aim to keep the `SuperchainERC20` contract as unopinionated as possible.

**Cantina Managed:** Acknowledged.

#### 5.5.6 Code quality improvements

**Severity:** Informational

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

**Description and Recommendations:** In order to improve code consistency, readability, and technical accuracy of the documentation the following changes are recommended:

1. Inconsistent Storage Slot Derivations: Storage slot namespace id and computation should follow a consistent pattern by adhering to EIP-7201 and using snake case namespace derivation:

```
// Current inconsistent approach:
bytes32 public constant PAUSED_SLOT = bytes32(uint256(keccak256("superchainConfig.paused")) - 1);
/// bytes32(uint256(keccak256("l2tol2crossdomainmessenger.sender")) - 1)
bytes32 internal constant CROSS_DOMAIN_MESSAGE_SENDER_SLOT =
0xb83444d07072b122e2e72a669ce32857d892345c19856f4e7142d06a167ab3f3;
/// keccak256(abi.encode(uint256(keccak256("l1Block.identifier.isDeposit")) - 1)) &
↪  ~bytes32(uint256(0xff))
uint256 internal constant IS_DEPOSIT_SLOT =
↪  0x921bd3a089295c6e5540e8fba8195448d253efd6f2e3e495b499b627dc36a300;

// Recommended consistent approach:
bytes32 public constant PAUSED_SLOT =
    keccak256(abi.encode(uint256(keccak256("superchainConfig.paused")) - 1)) &
    ↪  ~bytes32(uint256(0xff));
bytes32 internal constant CROSS_DOMAIN_MESSAGE_SENDER_SLOT =
    keccak256(abi.encode(uint256(keccak256("l2ToL2CrossDomainMessenger.paused")) - 1)) &
    ↪  ~bytes32(uint256(0xff));
bytes32 internal constant IS_DEPOSIT_SLOT =
    keccak256(abi.encode(uint256(keccak256("l1BlockInterop.isDeposit")) - 1)) &
    ↪  ~bytes32(uint256(0xff));
```

2. Documentation Typos and Inconsistencies:
   - Fix typo in `SuperchainWETH` natspec:

```
- /// @notice SuperchainWETH is a version of WETH that can be freely transfrered between
↪  chains
+ /// @notice SuperchainWETH is a version of WETH that can be freely transferred between
↪  chains
```

   - Maintain consistent terminology in L1BlockInterop:

```
- /// @notice Updates the `isDeposit` flag and sets the L1 block values for an Interop
↪  upgraded chain.
+ /// @notice Updates the isDeposit flag and sets the L1 block values for an Interop
↪  upgraded chain.
```

3. `CrossL2Inbox` Documentation Improvements:
   - Clarify validation scope:

```
- /// @notice Validates a cross chain message on the destination chain
-           and emits an ExecutingMessage event. This function is useful
-           for applications that understand the schema of the _message payload and want
↪  to
-           process it in a custom way.
+ /// @notice Validates that a cross-chain message (identified by Identifier) was emitted
↪  on any
+           chain in the dependency set and emits an ExecutingMessage event. This function
↪  is useful
+           for applications that understand the schema of the message payload and want to
+           process it in a custom way.
```

   - Clarify message structure:

```
- /// @param _msgHash Hash of the message payload to call target with.
+ /// @param _msgHash Hash of the message payload which corresponds to an encoded event in
↪  the format:
+                     event.selector || ...topics || ...data
```

4. `ETHLiquidity` Documentation Improvements:
   - Clarify deployment details:

```
- /// @notice The ETHLiquidity contract allows other contracts to access ETH liquidity
↪  without
-            needing to modify the EVM to generate new ETH.
+ /// @notice The ETHLiquidity contract allows other contracts to access ETH liquidity
↪  without
+            needing to modify the EVM to generate new ETH. Contract comes "pre-loaded"
+            with uint248.max balance to prevent liquidity shortages
```

5. Incorrect Interface Type in `OptimismPortalInterop`:
   - The initialize function uses the wrong interface type:

```
function initialize(
    IDisputeGameFactory _disputeGameFactory,
    ISystemConfig _systemConfig,
-     ISuperchainConfig _superchainConfig,
+     ISuperchainConfigInterop _superchainConfig,
    GameType _initialRespectedGameType
)
```

6. `L1BlockInterop` Documentation Improvements:
   - The natspec for deposit context handling should be expanded to better explain the behavior and constraints:

```
/// @notice L1BlockInterop manages deposit contexts within L2 blocks. A deposit context
///         represents a series of deposited transactions within a single block, starting
///         with an L1 attributes transaction and ending after the final deposit.
///         The expected sequence of operations in a deposit context is:
///         1. L1 attributes transaction opens the deposit context (isDeposit = true)
///         2. User deposits execute (if any exist)
///         3. L1 attributes transaction closes the deposit context (isDeposit = false)
///         Note: During upgrades, additional deposits may follow after this sequence.
contract L1BlockInterop is L1Block {
    // ...

    /// @notice Resets the isDeposit flag, marking the end of a deposit context.
    /// @dev Should only be called by the depositor account after all deposits are
    ↪  complete.
    ///      Chain operators must ensure ~9M gas buffer between guaranteed gas limit and
    ↪  L2
    ///      gas limit to safely destroy deposit contexts.
    function depositsComplete() external {
        if (msg.sender != DEPOSITOR_ACCOUNT()) revert NotDepositor();

        // Set the isDeposit flag to false.
        assembly {
            sstore(IS_DEPOSIT_SLOT, 0)
        }
    }
}
```

**OP Labs:** Commit ce783f9 addresses the following points:

1. Acknowledged.
2. Typo fixed and consistent terminology applied.
3. Acknowledged.
4. Natspec updated as recommended.
5. Acknowledged.
6. Natspec updated as recommended.

**Cantina Managed:** Addressed.

#### 5.5.7 Spamming the node with transactions that would be removed from the block

**Severity:** Informational

**Context:** CrossL2Inbox.sol#L49-L54

**Description:** `CrossL2Inbox.validateMessage()` is used to check that provided event was present on specific block of specific chain. In case the supervisor detects that event was not fired on source chain, then the sequencer will exclude such transactions from the block. In this case initiator of the transaction doesn't pay for the gas. This makes attack on the node possible, where the node is spammed with enormous amount of invalid transactions, which would consume computational power of the node. This issue is reported as informational, as the team is aware of the issue and already working on the solution.

**Recommendation:** It is recommended to implement filtering system that would detect and exclude malicious messages from processing by node.

**OP Labs:** Acknowledged. This will be addressed in a redesign of `CrossL2Inbox` prior to release.

**Cantina Managed:** Acknowledged.

### 5.5.8 Improve validation and error messages in payload decoding

**Severity:** Informational

**Context:** L2ToL2CrossDomainMessenger.sol#L242-L243

**Description:** The `_decodeSentMessagePayload` function in `L2ToL2CrossDomainMessenger` lacks robust input validation and returns unclear error messages when decoding fails. It also does not validate payload length, potentially accepting trailing bytes that will be ignored. This is not a security concern in its current form, as additional trailing bytes will result in different message hashes that cannot be properly validated.

**Recommendation:** Consider adding validation checks with clear error messages and length validation if desired. However, this change does increase execution cost.

**OP Labs:** Acknowledged. For now, we choose not to modify that logic, prioritizing a lesser gas cost over readability.

**Cantina Managed:** Acknowledged.