



# Audit Report March, 2025



# Table of Content

Executive Summary..... 02

Number of Security Issues per Severity ..... 03

Checked Vulnerabilities..... 04

Techniques and Methods..... 05

Types of Severity ..... 06

Types of Issues ..... 06

**Low Issues** ..... 07

    1. CEI Pattern Violation in `claim` Function

    2. Deprecation of ZERO\_B256 Constant in the Contract

    3. Incorrect Handling of msg\_amount() in Mailbox::dispatch ..... 07

Automated Tests..... 09

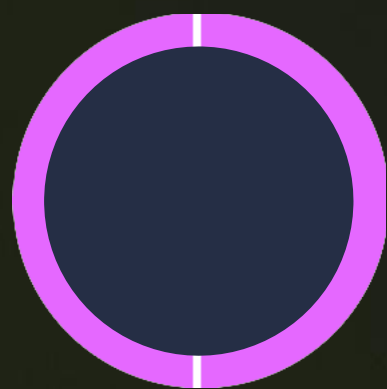
Closing Summary..... 10

Disclaimer ..... 10

# Executive Summary

Project Name	Hyperlane-fuel
Project URL	
Overview	Hyperlane is a permissionless interoperability protocol for cross-chain communication. It enables message passing and asset transfers across different chains without relying on centralized intermediaries or requiring any permissions.
Audit Scope	<a href="https://github.com/fuel-infrastructure/fuel-hyperlane-integration/commits/master/">https://github.com/fuel-infrastructure/fuel-hyperlane-integration/commits/master/</a>
Commit	941effc8bb46e3704cc4f947d81ef6ccdd5e4183
Language	Sway
Blockchain	Fuel
Method	Manual Analysis, Functional Testing, Automated Testing
First Review	1st March 2025 - 10th March 2025
Updated Code Received	10th March 2025
Second Review	10th March 2025

# Number of Security Issues per Severity



High

Medium

Low

Informational

Issues Found



High

Medium

Low

Informational

Open Issues

0

0

0

0

Acknowledged Issues

0

0

0

0

Partially Resolved Issues

0

0

0

0

Resolved Issues

0

0

3

0

# Checked Vulnerabilities

- ✓ Arbitrary write to storage
- ✓ Centralization of control
- ✓ Ether theft
- ✓ Improper or missing events
- ✓ Logical issues and flaws
- ✓ Arithmetic Computations Correctness
- ✓ Race conditions/front running
- ✓ Re-entrancy
- ✓ Malicious libraries
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ ERC's conformance
- ✓ Missing Zero Address Validation
- ✓ Revert/require functions
- ✓ Upgradeable safety
- ✓ Using inline assembly
- ✓ Style guide violation
- ✓ Parallel Execution safety
- ✓ UTXO Model Verification
- ✓ FuelVM Opcodes
- ✓ Cross-Chain Interactions
- ✓ Modular Design
- ✓ Access Control Vulnerabilities
- ✓ Denial of Service (DoS)
- ✓ Oracle Manipulation
- ✓ Signature Replay Attacks
- ✓ Improper Handling of External Calls
- ✓ Proxy Storage Collision
- ✓ Use of Deprecated Functions

# Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

## Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

## Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

## Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

## Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

## Tools and Platforms used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity statistic analysis.

## Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

### High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

### Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

### Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

### Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

## Types of Issues

### Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

### Resolved

These are the issues identified in the initial audit and have been successfully fixed.

### Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

### Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

# Issue Details

## L-01 CEI Pattern Violation in claim Function

Severity: Low

### Description

The claim function in the contract does not follow the Checks-Effects-Interactions (CEI) pattern, which is a best practice for secure smart contract design. The function performs an external call (transfer) before updating storage, which could lead to minor inconsistencies in contract state. However, since the function is restricted to only the owner, the risk of reentrancy is none.

### Affected Code

#### Function: claim

```
fn claim(asset: Option<AssetId>) {  
  
    only_owner();  
  
    let beneficiary = storage.beneficiary.read();  
  
    let stored_asset = storage.asset_id.read();  
  
    let asset = asset.unwrap_or(stored_asset);  
  
    let balance = this_balance(asset);  
  
    transfer(beneficiary, asset, balance); // @audit Transfer before storage update  
  
    if stored_asset == asset {  
  
        storage.contract_balance.write(0);  
  
    }  
  
    log(ClaimEvent {  
  
        beneficiary,  
  
        amount: balance,  
  
    });
```



## Impact

- **Minimal Security Risk:** The function is **restricted to the owner**, so reentrancy or malicious behavior is unlikely. However, following best practices ensures long-term security.

## Recommended Fix

Follow the CEI pattern by updating storage before transferring funds.

### Fixed Code:

```
fn claim(asset: Option<AssetId>) {  
  
    only_owner();  
  
    let beneficiary = storage.beneficiary.read();  
  
    let stored_asset = storage.asset_id.read();  
  
    let asset = asset.unwrap_or(stored_asset);  
  
    let balance = this_balance(asset);  
  
    // Update storage before external call  
  
    if stored_asset == asset {  
  
        storage.contract_balance.write(0);  
  
    }  
  
    // Perform the external call after storage update  
  
    transfer(beneficiary, asset, balance);  
  
    log(ClaimEvent {beneficiary,  
  
        amount: balance,  
  
    });
```

## L-02 Deprecated Use of ZERO\_B256

**Severity:** Low

### Description

The contract makes use of constants::`ZERO_B256`, which is now deprecated according to the updated **Sway language standards**. Using deprecated constants may lead to **future compatibility issues** as the language evolves. The recommended approach is to use `b256::zero()` instead, which aligns with the latest standards

## Affected Code

```
let zero_hash = constants::ZERO_B256; // @audit Deprecated usage
```

## Impact

- **Future Compatibility Issues:** The use of ZERO\_B256 may result in warnings or errors in future versions of the Sway compiler.
- **Code Maintainability:** Keeping the contract up to date with the latest Sway standards ensures long-term maintainability and easier debugging.

## Recommendation

Replace all occurrences of constants::ZERO\_B256 with b256::zero() to comply with updated Sway language standards.

### Fixed Code:

```
let zero_hash = b256::zero(); // Updated as per latest Sway standards
```

---

## L-03 Incorrect Handling of msg\_amount() in Mailbox::dispatch

**Severity:** Low

## Description

In the Mailbox::dispatch function (line 271), there is a check:

```
if (msg_amount() < required_value) { required_value = msg_amount(); }
```

This logic sets required\_value to msg\_amount() if the sent amount is less than the required amount. However, this can lead to unintended reverts during external contract interactions, as the function will proceed with a lower value than required.

To prevent unnecessary contract execution, it is recommended to revert immediately if msg\_amount() is insufficient, rather than adjusting required\_value downward.

## Impact

- **Unnecessary External Calls:** The function proceeds with an incorrect required\_value, which ultimately causes reverts later in the execution.
- **Gas Inefficiency:** Wasted computation and failed transactions could increase user costs

## Recommendation

Instead of modifying required\_value, the function should revert immediately when msg\_amount() is insufficient.

```
if (msg_amount() < required_value) { revert("Insufficient funds sent for operation") }
```

# Closing Summary

In this report, we have considered the security of Hyperlane . We performed our audit according to the procedure described above.

Some issues of informational severity were found, which the Hyperlane Team has Fixed.

## Disclaimer

SafeEdges Smart contract security audit provides services to help identify and mitigate potential security risks in Hyperlane . However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. SafeEdges audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Hyperlane. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

SafeEdges cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of Hyperlane to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

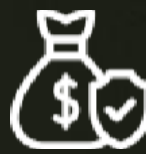
# About **SAFE EDGES**

SafeEdges is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over \$30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.



**500+**

Audits Completed



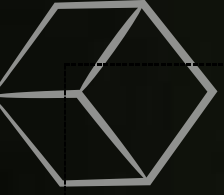
**\$3B**

Secured



**600k+**

Lines of Code Audited





# Audit Report

## March, 2025



**Safe Edges**

 <https://safeedges.in>

 [info@safeedges.in](mailto:info@safeedges.in)