MiloTruck

LUKSO

Security Review

August 27, 2025

Contents

1 Introduction

1.1 About MiloTruck

MiloTruck is an independent security researcher, primarily working as a Lead Security Researcher at Spearbit and Cantina. Previously, he was part of the team at Renascence Labs and a Lead Auditor at Trust Security.

For private audits or security consulting, please reach out to him on Twitter *@milotruck*.

1.2 Disclaimer

A smart contract security review **can never prove the complete absence of vulnerabilities**. Security reviews are a time, resource and expertise bound effort to find as many vulnerabilities as possible. However, they cannot guarantee the absolute security of the protocol in any way.

2 Risk Classification

Severity Level	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol's core functionality.
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol's functionality/availability.
- Low - Funds are **not** at risk.

2.2 Likelihood

- High - Highly likely to occur.
- Medium - Might occur under specific conditions.
- Low - Unlikely to occur.

3 Executive Summary

3.1 About LUKSO

LUKSO is the digital base layer for the New Creative Economies. It provides creators and users with future-proof tools and standards to unleash their creative force in an open interoperable ecosystem.

3.2 Overview

Project Name	LUKSO
Project Type	Bridge
Language	Solidity
Repository	lukso-network/lsp-bridge-HypLSP7
Commit Hash	lsp-bridge-HypLSP7

3.3 Issues Found

High	0
Medium	0
Low	2
Informational	2

4 Findings

4.1 Low Risk

4.1.1 `MovableCollateralRouter.approveTokenForBridge()` is incompatible with LSP7

Context: MovableCollateralRouter.sol#L91-L96

Description: Hyperlane's `MovableCollateralRouter` contract includes a `approveTokenForBridge()` function, which calls `approve()`:

```
function approveTokenForBridge(
    IERC20 token,
    ValueTransferBridge bridge
) external onlyOwner {
    token.safeApprove(address(bridge), type(uint256).max);
}
```

This is incompatible with LSP7, which does not have an `approve()` function.

Recommendation: Add a new function which calls `authorizeOperator()` instead.

LUKSO: Fixed in PR 29.

MiloTruck: Verified, the recommendation has been implemented.

4.1.2 Recipient addresses with dirty upper bytes can never be processed

Context:

- TokenRouter.sol#L237-L241
- TypeCasts.sol#L11-L17
- TokenRouter.sol#L54-L58

Description: In Hyperlane's `TokenRouter` contract, `_handle()` converts `recipient` from `bytes32` to an `address` via `bytes32ToAddress()`:

```
_transferTo(
    recipient.bytes32ToAddress(),
    _inboundAmount(amount),
    metadata
);
```

`bytes32ToAddress()` includes a check which ensures the upper 12 bytes of `recipient` are empty:

```
function bytes32ToAddress(bytes32 _buf) internal pure returns (address) {
    require(
        uint256(_buf) <= uint256(type(uint160).max),
        "TypeCasts: bytes32ToAddress overflow"
    );
    return address(uint160(uint256(_buf)));
}
```

However, `transferRemote()` takes in the `_recipient` as a `bytes32`:

```
function transferRemote(
    uint32 _destination,
    bytes32 _recipient,
    uint256 _amountOrId
) external payable virtual returns (bytes32 messageId) {
```

This means it is possible for the upper 12 bytes of `_recipient` to be non-zero. Therefore, if `transferRemote()` is wrongly called with dirty upper bytes, calling `handle()` on the destination chain will always revert. As a result, tokens will be lost since the message can never be processed on the destination chain.

Recommendation: Considering overriding `transferRemote()` to check that the upper 12 bytes of `_recipient` are empty.

Alternatively, include a warning which states `transferRemote()` should never be called with dirty upper bytes.

LUKSO: Acknowledged.

MiloTruck: Acknowledged.

4.2 Informational

4.2.1 `HypLSP7Collateral.rebalance()` allows a rebalancer to mint infinite tokens

Context: MovableCollateralRouter.sol#L158-L169

Description: In Hyperlane's `MovableCollateralRouter` contract, `_rebalance()` calls `transferRemote()` for the specified `bridge` address:

```
function _rebalance(
    uint32 domain,
    bytes32 recipient,
    uint256 amount,
    ValueTransferBridge bridge
) internal virtual {
    bridge.transferRemote{value: msg.value}({
        destinationDomain: domain,
        recipient: recipient,
        amountOut: amount
    });
}
```

This transfers tokens from the `MovableCollateralRouter` contract to the bridge and sends a cross-chain message to the destination chain.

However, if the `bridge` address is allowed to be the `MovableCollateralRouter` contract itself, a rebalancer will be able to mint infinite tokens on the destination chain. For example:

- Assume a lock-and-mint bridge for USDC is setup as follows:
  - Ethereum has a `HypERC20Collateral` contract, which locks USDC.
  - LUKSO has a `HypLSP7` contract, which mints synthetic USDC.
- Also, assume the `HypERC20Collateral` on Ethereum holds some USDC (e.g. 10k USDC already locked).
- On Ethereum, the rebalancer calls `rebalance()` with the `bridge` parameter as the `HypERC20Collateral` contract address itself (that contains the locked USDC):
  - `_rebalance()` self-calls the `transferRemote()` function.
  - `transferRemote()` eventually calls `USDC.transfer()` with both `from` and `to` as the `HypERC20Collateral` contract.
  - A cross-chain message is sent to the destination chain, which mints synthetic USDC on LUKSO.

As seen from above, since `transferRemote()` calls `transfer()` with `from == to`, the balance of the contract does not change. Therefore, a rebalancer can repeatedly call `rebalance()` to mint infinite synthetic USDC on the destination chain.

Note that this requires a malicious owner to whitelist the `MovableCollateralRouter` contract address as an external bridge via `addBridge()`.

LUKSO: Acknowledged.

MiloTruck: Acknowledged.

4.2.2 Authorized operators cannot call `transferRemote()` in `HypLSP8`

Context: HypLSP8.sol#L75-L80

Description: `HypLSP8._transferFromSender()` checks that `msg.sender` is the owner of `tokenId`:

```
function _transferFromSender(uint256 tokenId) internal virtual override returns (bytes memory) {
    bytes32 tokenIdAsBytes32 = bytes32(tokenId);
    require(tokenOwnerOf(tokenIdAsBytes32) == msg.sender, "!owner");
    _burn(tokenIdAsBytes32, "");
    return bytes(""); // no metadata
}
```

As such, authorized operators will not be able to call `transferRemote()`. This behavior is inconsistent with `HypLSP8Collateral`, which calls `transfer()` in its `_transferFromSender()` function and allows authorized operators to bridge on the owner's behalf.

Recommendation: Consider calling `_isOperatorOrOwner()` instead:

```
  function _transferFromSender(uint256 tokenId) internal virtual override returns (bytes memory) {
      bytes32 tokenIdAsBytes32 = bytes32(tokenId);
-     require(tokenOwnerOf(tokenIdAsBytes32) == msg.sender, "!owner");
+     require(_isOperatorOrOwner(msg.sender, tokenIdAsBytes32), "!owner");
      _burn(tokenIdAsBytes32, "");
      return bytes(""); // no metadata
  }
```

LUKSO: Acknowledged.

MiloTruck: Acknowledged.