Pashov Audit Group

Hyperlend Security Review

November 21st 2025 - November 22nd 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Hyperlend Utils

Hyperlend Utils contracts provide a set of tools for managing user positions on Hyperlend, including automated liquidation protection, collateral swaps, and batch borrowing/repayment, while integrating safeguards like Health Factor thresholds, TWAP checks, and per-token caps. They are designed to simplify interactions with HyperLend and HyperCore by offering read-only margin tracking, atomic operations, and configurable user protections without relying heavily on external CoreWriter actions.

5. Executive Summary

A time-boxed security review of the **hyperlendx/hyperlend-utils** and **hyperlendx/hyperevm-oracle** repositories was done by Pashov Audit Group, during which **t.aksoy**, **DemoreXTess**, **Drynooo** engaged to review **Hyperlend Utils**. A total of **6** issues were uncovered.

Protocol Summary	
Project Name	Hyperlend Utils
Protocol Type	Lending Tools
Timeline	November 21st 2025 - November 22nd 2025

Review commit hashes:

- 835056165de225941e949beea66747e9df9d2164 (hyperlendx/hyperlend-utils)
- e352f15eba26dd24748617664ca044572c821a75 (hyperlendx/hyperevm-oracle)

Fixes review commit hashes:

- 5b58ed59e39e407789e3bc54013f6f466f991b7c (hyperlendx/hyperlend-utils)
- ebb2fd00ce6737d984c3a94585244ad7984bef1c (hyperlendx/hyperevm-oracle)

Scope

`UtaHelper.sol` `DualFallbackOracle.sol`

6. Findings

Findings count

Severity	Amount
High	1
Low	5
Total findings	**6**

Summary of findings

ID	Title	Severity	Status
[H-01]	Deprecated `safeApprove()` usage blocks collateral approval to pool	High	Resolved
[L-01]	Race condition on core rescue	Low	Resolved
[L-02]	Fallback logic fails to choose fresher oracle	Low	Resolved
[L-03]	Oracle lacks emergency price validation	Low	Resolved
[L-04]	Missing try/catch in oracle calls disables fallback	Low	Resolved
[L-05]	Deployment fails if optional `Emergency_Source` is address(0)	Low	Resolved

High findings

[H-01] Deprecated `safeApprove()` usage blocks collateral approval to pool

Severity

Impact: Medium

Likelihood: High

Description

Project uses Openzeppelin@v4.9.6, which deprecates `safeApprove()`. Safe approve usage is not safe here because it blocks every approval if approval for the collateral is non-zero:

```
//    function executeOperation()

        for (uint256 i = 0; i < _collateralActions.length; ++i){
            uint256 amount = _collateralActions[i].amount;
            IERC20 token = _collateralActions[i].token;

            //transfer tokens from the caller & approve pool contract to spend them
            token.safeTransferFrom(msg.sender, address(this), amount);
@>          token.safeApprove(address(pool), type(uint256).max);

            //supply tokens on behalf of the msg.sender
            pool.supply(address(token), amount, msg.sender, 0);
        }
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/dc44c9f1a4c3b10af99492eed84f83ed244203f6/contracts/token/ERC20/utils/SafeERC20.sol#L45-L54

```
    function safeApprove(IERC20 token, address spender, uint256 value) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0) || (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender,
value));
    }
```

After the first approval for collateral, every other call will fail because of the `require` line in `safeApprove()`.

Recommendations

Consider using `forceApprove()` instead of `safeApprove()`.

Low findings

[L-01] Race condition on core rescue

If owner calls rescueHyperCore() for USDC token while users are executing borrow, the rescue can drain funds before the user's transfer executes on HyperCore, causing silent fund loss. According to HyperEVM specs, CoreWriter actions execute after USDC transfers arrive. If both owner rescue and user transfer happen in the same block user transfer can fail silently.

Block Execution Sequence (per HyperEVM spec)

```
Step 1: L1 block is built
Step 2: EVM block is built (both transactions execute here)
Step 3: EVM -> Core transfers are processed
Step 4: CoreWriter actions are processed
```

USDC rescues, and user borrows shouldn't be allowed in same blocks. If this is allowed, USDC rescues should be carefully handled.

[L-02] Fallback logic fails to choose fresher oracle

When the primary oracle is unhealthy, the contract checks the fallback oracle. If fallback is also unhealthy → the contract returns the primary oracle data, even if fallback is less stale. System may return older or more inaccurate data.

If both are unhealthy, choosing the latest price could be a better logic.

[L-03] Oracle lacks emergency price validation

When `isEmergencyOracleEnabled = true,` the contract immediately returns the emergency oracle's latestRoundData(), regardless of data freshness or price correctness. Even though it may be zero-priced, or return invalid data.

Add a price check for the emergency oracle, and if it is not valid, other oracle prices should be returned.

[L-04] Missing try/catch in oracle calls disables fallback

The `DualFallbackOracle` contract is designed to utilize a fallback oracle if the primary source is unhealthy. However, the contract performs direct external calls to `PRIMARY_SOURCE.latestRoundData()` and `FALLBACK_SOURCE.latestRoundData()`.

If the primary oracle reverts (for example, if the underlying aggregator is paused or encounters an internal error), the entire transaction will revert. The execution flow will never reach the logic that checks `_isPrimaryHealthy` or attempts to query the fallback source, effectively rendering the redundancy mechanism useless.

```
// hyperevm-oracle/contracts/adapters/DualFallbackOracle.sol

(
    uint80 _roundId,
    int256 _answer,
    uint256 _startedAt,
    uint256 _updatedAt,
    uint80 _answeredInRound
) = PRIMARY_SOURCE.latestRoundData(); // @audit If this reverts, the whole tx fails
```

Recommendations

Wrap the calls to `latestRoundData` in a `try/catch` block. If the primary source call fails, the catch block should handle the error by attempting to fetch data from the fallback source.

[L-05] Deployment fails if optional `Emergency_Source` is address(0)

The logic in `getData` suggests that `EMERGENCY_SOURCE` is optional, as it explicitly checks if the address is not zero before using it. However, the constructor unconditionally calls `decimals()` on `EMERGENCY_SOURCE` for validation.

If a deployer attempts to initialize the contract with `address(0)` as the `_emergencySource` (intending to set it later or leave it disabled), the deployment will revert because calling `decimals()` on the zero address fails.

```
        if (PRIMARY_SOURCE.decimals() != FALLBACK_SOURCE.decimals()) revert InvalidDecimals();
        // @audit Reverts if EMERGENCY_SOURCE is address(0)
        if (PRIMARY_SOURCE.decimals() != EMERGENCY_SOURCE.decimals()) revert InvalidDecimals();
        decimals = PRIMARY_SOURCE.decimals();
```

Recommendations

Add a check to ensure `EMERGENCY_SOURCE` is not `address(0)` before validating decimals in the constructor.