Hyperlend Security Review

Pashov Audit Group

Conducted by: 0xunforgiven, Hals, MrPotatoMagic

January 11th 2025 - January 17th 2025

Contents

1. About Pashov Audit Group 2

2. Disclaimer 2

3. Introduction 2

4. About Hyperlend 3

5. Risk Classification 3

5.1. Impact 3

5.2. Likelihood 4

5.3. Action required for severity levels 4

6. Security Assessment Summary 5

7. Executive Summary 7

8. Findings 9

8.1. Medium Findings 9

[M-01] StrategyManager functions are not payable 9

[M-02] Looping.openPosition() insufficient division factor 10

[M-03] openPosition() DoS when _initialAmount is _yieldAsset 11

[M-04] Unrevoked approvals causing swap failures in UniswapV3Swapper.swap() 14

8.2. Low Findings 16

[L-01] Code does not verify the path Looping 16

[L-02] Incorrect check to close the user's position 16

[L-03] Swap fee is hardcoded in UniswapV3Swapper.swap() 18

[L-04] Code does not allow users to specify a deadline for swaps 19

[L-05] StrategyManager.executeCall() incorrect handling of allowRevert 20

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check 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. Introduction

A time-boxed security review of the **hyperlendx/hyperlend-core, hyperlendx/hyperlend-isolated, hyperlendx/core-config-engine, hyperlendx/looping-contracts, hyperlendx/cross-chain-lending-deposits** repositories was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Hyperlend

HyperLend is a lending platform built on Hyperliquid's EVM blockchain, allowing users to supply assets to liquidity pools and borrow against their deposits. It utilizes algorithmic interest rate models based on supply and demand, ensuring efficient capital utilization. The platform supports flashloans, peer-to-peer lending, isolated and core pools.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. 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.

5.3. 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

6. Security Assessment Summary

review commit hashes:

- 4256249f94b8762a5ce41caa2283c0d989efc593
- 37c678450f37a923517e7a203f8353599e143b7e
- 0339f192bb98b55c856d1cc6f76a04bd6fe687ee
- 0fdde7b1033263b7c7579c9ec505dfb635197f3c
- 2576caa13409d49bd886965d3c4b3fe61e40397e

fixes review commit hashes:

- c8ea750400e62ef089ed95ea2851afe7027646e5
- a496d810a66df9b889306b7e6807ba354785d59e

Scope

The following smart contracts were in scope of the audit:

- `HyperlendPair`
- `HyperlendPairAccessControl`
- `HyperlendPairAccessControlErrors`
- `HyperlendPairDeployer`
- `HyperlendPairRegistry`
- `HyperlendWhitelist`
- `ReservesSetupHelper`
- `SeedAmountsHolder`
- `Oracle`
- `ProtocolDataProvider`
- `PriceOracleSentinel`
- `FlashLoanLogic`
- `DataTypes`
- `Pool`
- `AToken`
- `StableDebtToken`
- `ACLConfigEngine`
- `CapsConfigEngine`
- `ConfiguratorInputTypes`
- `DataTypes`
- `Context`
- `Ownable`
- `ListingConfigEngine`
- `ListingsConfigEngineFactory`
- `UiDataProvider`
- `Looping`
- `StrategyManager`
- `StrategyManagerFactory`
- `UniswapV3Swapper`

7. Executive Summary

Over the course of the security review, 0xunforgiven, Hals, MrPotatoMagic engaged with Hyperlend to review Hyperlend. In this period of time a total of **9** issues were uncovered.

Protocol Summary

Protocol Name	Hyperlend
Repository	https://github.com/hyperlendx/hyperlend-core
Date	January 11th 2025 - January 17th 2025
Protocol Type	Lending

Findings Count

Severity	**Amount**
Medium	4
Low	5
Total Findings	**9**

Summary of Findings

ID	Title	Severity	Status
[M-01]	StrategyManager functions are not payable	Medium	Resolved
[M-02]	Looping.openPosition() insufficient division factor	Medium	Resolved
[M-03]	openPosition() DoS when _initialAmount is _yieldAsset	Medium	Resolved
[M-04]	Unrevoked approvals causing swap failures in UniswapV3Swapper.swap()	Medium	Resolved
[L-01]	Code does not verify the path Looping	Low	Acknowledged
[L-02]	Incorrect check to close the user's position	Low	Resolved
[L-03]	Swap fee is hardcoded in UniswapV3Swapper.swap()	Low	Resolved
[L-04]	Code does not allow users to specify a deadline for swaps	Low	Resolved
[L-05]	StrategyManager.executeCall() incorrect handling of allowRevert	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] StrategyManager functions are not `payable`

Severity

Impact: Low

Likelihood: High

Description

Users can use StrategyManager to perform their strategies in one transaction and the code allows sending ETH for the transactions:

```
function executeCall(
      addresstarget,
      uint256value,
      bytesmemorydata,
      boolallowRevert
    ) public onlyOwner(
        (bool success, bytes memory returnData) = target.call{value: value}
          (data);
        if (!allowRevert) require(success, 'execution reverted');
        return returnData;
    }
```

The issue is that none of the functions of the StrategyManager are payable and there's no payable `fallback()` or `receive()` function so there's no way to transfer ETH to StrategyManager address and perform transactions that require sending ETH.

Recommendations

Mark StrategyManager functions as `payable`.

[M-02] `Looping.openPosition()` insufficient division factor

Severity

Impact: Medium

Likelihood: Medium

Description

In `Looping.openPosition()` function, when users provide the expected flashloan premium payment in the `yieldAsset`, it is swapped into `debtAsset`, where the `minAmountOut` expected from the swap is calculated as follows:

```
function openPosition(
        address _pool,
        address _swapper,
        address _debtAsset,
        address _yieldAsset,
        uint256 _initialAmount,
        uint256 _flashloanAmount,
        uint256 _minAmountOut,
        address[] memory _path,
        bool _startWithYield
    ) external nonReentrant {
        //...

        if (_startWithYield) {
         //...
            uint256 minAmountOut = (((_flashloanAmount * 1e8) / _minAmountOut) *
                _initialAmount) / 1e8;
            //...
        } else {
            //...
        }
        //...
    }
```

The `1e8` factor is introduced to account for rounding down issues when dividing by the `_minAmountOut`.

However, if the `yieldAsset decimals` > `debtAsset decimals + 8`; the introduced factor will not provide sufficient protection against rounding down, which could result in the calculation of a smaller `minAmountOut` value than expected.

Recommendations

Consider increasing the `1e8` factor to a higher value (`1e36` for example) to ensure better precision handling:

```
function openPosition(
        address _pool,
        address _swapper,
        address _debtAsset,
        address _yieldAsset,
        uint256 _initialAmount,
        uint256 _flashloanAmount,
        uint256 _minAmountOut,
        address[] memory _path,
        bool _startWithYield,
+       uint256 _minInitialAmountOut
    ) external nonReentrant {
        //...

        if (_startWithYield) {
            //...
-           uint256 minAmountOut = (((_flashloanAmount * 1e8) / _minAmountOut) *
-               _initialAmount) / 1e8;

+           uint256 minAmountOut = ((
+ (_flashloanAmount * 1e36) / _minAmountOut) *
+               _initialAmount) / 1e36;

        //...
    }
```

[M-03] `openPosition()` DoS when `_initialAmount` is `_yieldAsset`

Severity

Impact: Medium

Likelihood: Medium

Description

- The `Looping.openPosition()` function enables users to provide a flashloan premium payment (`_initialAmount`) either in `yieldAsset` or in `debtAsset`:

```
function openPosition(
        address _pool,
        address _swapper,
        address _debtAsset,
        address _yieldAsset,
        uint256 _initialAmount,
        uint256 _flashloanAmount,
        uint256 _minAmountOut,
        address[] memory _path,
        bool _startWithYield
    ) external nonReentrant {
        require(pools[_pool], "pool not allowed");

        if (_startWithYield) {
            //transfer initial _yieldAsset from user
            IERC20(_yieldAsset).transferFrom(
                msg.sender,
                address(this),
                _initialAmount
            );
            uint256 minAmountOut = (((_flashloanAmount * 1e8) / _minAmountOut) *
                _initialAmount) / 1e8;
            _initialAmount = _swap(
                _swapper,
                _reversePath(_path),
                _initialAmount,
                minAmountOut
            );
        } else {
            //...
        }
        //...
    }
```

- If the `_initialAmount` is in `yieldAsset`, then a swap is done to get the `_initialAmount` in `debtAsset`, where the `minAmountOut` expected from the swap is calculated as follows:

```
uint256 minAmountOut = ((
  (_flashloanAmount * 1e8) / _minAmountOut) * _initialAmount) / 1e8;
```

Where in this case:

- `_flashloanAmount` : the amount of the `_debtAsset` to be flashloaned then swapped to `_yieldAsset`
- `_minAmountOut` : minimum amount of `_yieldAsset` expected to be received after swapping `_debtAsset`
- `_initialAmount` : the initial amount of the `_yieldAsset` provided by the user
- But as can be noticed, the `minAmountOut` calculation assumes that both `_debtAsset` and `_yieldAsset` have the same valuation, which is incorrect, for example:
  - a user flashloans `100_000 USDC`, while the `_minimumAmountOut` of WBTC to be received is `0.9995 WBTC` with 0.05% acceptable slippage (assuming the price of 1WBTC is 100_000USDC),
  - assuming that the flashloan premium is set to 0.1% of the flashloan amount, the user will provide 100USDC equivalent of WBTC to execute the `openPosition()`, which is `0.001` WBTC
  - so the calculated `minAmountOut` would be **100.050025 in USDC decimals** which is > the value of the provided `_initialAmount`:

```
minAmountOut = ((
  (100_000*1e6 * 1e8) / 0.9995*1e8) * 0.001*1e8) / 1e8 = 100.050025 in USDC decim
```

- This will result in calculating `minAmountOut` > `_initialAmount` when swapping, which will result in reverting the transaction.

Recommendations

Update the `Looping.OpenPosition()` function to accept this minimum amount as an argument:

```
function openPosition(
        address _pool,
        address _swapper,
        address _debtAsset,
        address _yieldAsset,
        uint256 _initialAmount,
        uint256 _flashloanAmount,
        uint256 _minAmountOut,
        address[] memory _path,
        bool _startWithYield,
+       uint256 _minInitialAmountOut
    ) external nonReentrant {
        require(pools[_pool], "pool not allowed");

        if (_startWithYield) {
            //transfer initial _yieldAsset from user
            IERC20(_yieldAsset).transferFrom(
                msg.sender,
                address(this),
                _initialAmount
            );
-           uint256 minAmountOut = (((_flashloanAmount * 1e8) / _minAmountOut) *
-               _initialAmount) / 1e8;
            _initialAmount = _swap(
                _swapper,
                _reversePath(_path),
                _initialAmount,
-               minAmountOut
+               _minInitialAmountOut
            );
        } else {
            //...
        }
        //...
    }
```

[M-04] Unrevoked approvals causing swap failures in `UniswapV3Swapper.swap()`

Severity

Impact: Medium

Likelihood: Medium

Description

The `UniswapV3Swapper.swap()` function has an issue where it does not revoke tokens approvals from the router after the swap, leading to a potential DoS with tokens like `USDT`, where this issue arises due to the fact that approvals are not reset after swaps, and if a token like `USDT` is used, modifying an allowance from a non-zero value to another non-zero value will revert if the allowance of the previous swap is not fully consumed by the router, which will result in reverting the swap transaction.

Similar issue was found with the different impact in the `Looping._swap()` function.

Recommendations

Update the `UniswapV3Swapper.swap()` function to revoke the tokens approval from the router after the swap:

```
function swap(
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        uint256 minAmountOut,
        address to,
        address referrer, // Unused in this implementation
        uint256 deadline
    ) external override returns (uint256 amountOut) {
        IERC20(tokenIn).transferFrom(msg.sender, address(this), amountIn);
        IERC20(tokenIn).approve(address(router), amountIn);
        //...
+       IERC20(tokenIn).approve(address(router), 0);
    }
```

8.2. Low Findings

[L-01] Code does not verify the path Looping

Whenever users call `openPosition()` or `closePosition()` they pass the `path` variable which is used to perform swaps. The issue is that the code doesn't verify the value of `path` to make sure that the first and last addresses in the path list are equal to the tokens that are going to be swapped. In some cases if the user specifies a wrong value for the last address in the path then the code would receive the wrong tokens and those tokens would stay in the contract address and the user would lose funds. It's better to verify the `path` parameter's value to avoid such edge cases.

[L-02] Incorrect check to close the user's position

The `Looping._executeClosePosition()` function has a check to close the user's position if `repaymentAmount == type(uint256).max`, however, `repaymentAmount` is decoded from the `params`, which represents the `flashloanAmount`, and since the `flashloanAmount` cannot be set to `type(uint256).max` (as it is not achievable), this check is redundant:

```
function _executeClosePosition(
        bytes memory params,
        address debtAsset
    ) internal {
        (
            ,
            //action type
            address yieldAsset,
            address swapper,
            address[] memory path,
            uint256 repaymentAmount, //=flashloanAmount
            uint256 minAmountOut,
            address user,
            uint256 withdrawAmount
        ) = abi.decode(
                params,
                (
                    uint8,
                    address,
                    address,
                    address[],
                    uint256,
                    uint256,
                    address,
                    uint256
                )
            );

        //...

        if (repaymentAmount == type(uint256).max) {
            repaymentAmount = debtDebtToken.balanceOf(user);
            withdrawAmount = hYieldToken.balanceOf(user);
        }

        //...
    }
```

Recommendations

Update the check to use `withdrawAmount` instead of `repaymentAmount`, as it can be set to `type(uint256).max`:

```
function _executeClosePosition(
        bytes memory params,
        address debtAsset
    ) internal {
        (
            ,
            //action type
            address yieldAsset,
            address swapper,
            address[] memory path,
            uint256 repaymentAmount, //=flashloanAmount
            uint256 minAmountOut,
            address user,
            uint256 withdrawAmount
        ) = abi.decode(
                params,
                (
                    uint8,
                    address,
                    address,
                    address[],
                    uint256,
                    uint256,
                    address,
                    uint256
                )
            );

        //...

-       if (repaymentAmount == type(uint256).max) {
+       if (withdrawAmount == type(uint256).max) {
           repaymentAmount = debtDebtToken.balanceOf(user);
           withdrawAmount = hYieldToken.balanceOf(user);
       }

        //...
    }
```

[L-03] Swap fee is hardcoded in `UniswapV3Swapper.swap()`

In `UniswapV3Swapper.swap()` function: a fixed swap fee of 0.3% (3000) is hardcoded when constructing the `ExactInputSingleParams`, however, Uniswap V3 typically offers several pools for a given token pair, each with different fee rates.

For example, swapping `USDC` for `WETH` is generally more efficient with a 0.05% (500) fee pool, which has better liquidity and less slippage compared to the 0.3% (3000) fee pool.

If the optimal pool is unavailable, the swap may fail:

```
function swap(
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        uint256 minAmountOut,
        address to,
        address referrer, // Unused in this implementation
        uint256 deadline
    ) external override returns (uint256 amountOut) {
       //...
        uint256 balanceBefore = IERC20(tokenOut).balanceOf(to);
        IUniswapV3Router.ExactInputSingleParams memory params = IUniswapV3Router
            .ExactInputSingleParams({
                tokenIn: tokenIn,
                tokenOut: tokenOut,
                fee: 3000,
                recipient: to,
                deadline: deadline,
                amountIn: amountIn,
                amountOutMinimum: minAmountOut,
                sqrtPriceLimitX96: 0
            });
       //...
    }
```

Recommendation: update the `UniswapV3Swapper.swap()` function to pass the fee as a parameter, allowing for dynamic selection of the optimal pool for each swap.

[L-04] Code does not allow users to specify a deadline for swaps

Users can open or close leveraged positions by using the Looping contract. Code performs swaps during the operation.

```
ISwapper(swapper).swapExactTokensForTokensSupportingFeeOnTransferTokens(
            amountToSwap,
            minAmountOut,
            path,
            address(this),
            referralAddress,
            block.timestamp
        );
```

The issue is that the code swaps the user's tokens and it uses `block.timestamp` for the swap's deadline and it doesn't allow the user to specify the deadline for swaps. As swap results may change if time passes, so it's a common practice to allow users to specify a deadline for their swaps to have more control over on-chain swap executions to avoid losses.

Allow users to specify the deadline and use it when calling swap.

[L-05] `StrategyManager.executeCall()` incorrect handling of `allowRevert`

The `StrategyManager.executeCall()` function is intended to be called by the owner to execute external calls, where it's primarily designed to allow users to manage their HyperLend positions:

```
function executeCall(
        address target,
        uint256 value,
        bytes memory data,
        bool allowRevert
    ) public onlyOwner returns (bytes memory) {
        (bool success, bytes memory returnData) = target.call{value: value}(
            data
        );
        if (!allowRevert) require(success, "execution reverted");
        return returnData;
    }
```

The function has the `allowRevert` parameter, where, if set to `true`, the failed call should **revert**, however, the current check implementation causes the call to revert when `allowRevert` is set to `false` when the owner doesn't intend for the call to revert, which contradicts the intended use of `allowRevert`.

As a result, if the owner executes a call and sets `allowRevert = true` (meaning the call should revert on failure), the transaction will not revert, and any changes made by the call will not be rolled back, where this could lead to a loss of the sent value and undesired results or state changes, especially if the call is part of a sequence of calls executed by the owner.

```
function executeCall(
        address target,
        uint256 value,
        bytes memory data,
        bool allowRevert
    ) public onlyOwner returns (bytes memory) {
        (bool success, bytes memory returnData) = target.call{value: value}(
            data
        );
-       if (!allowRevert) require(success, "execution reverted");
+       if (allowRevert) require(success, "execution reverted");

        return returnData;
    }
```