Blueberry Security Review

Pashov Audit Group

Conducted by: unforgiven, zark, aslanbek

May 16th 2025 - May 19th 2025

Contents

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 **Blueberryfi/blueberry-v2-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Blueberry

Blueberry is a leverage lending protocol that enables users to lend and borrow assets with up to 25x leverage, serving as DeFi's prime brokerage for executing diverse trading strategies. This second version of Blueberry is a complete rewrite, focusing on design improvements, and scalability to make decentralized leverage trading more accessible. This audit was focused on V2 of the Blueberry protocol.

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 hash - ac6bbab032bd36823e8762ee6ecbdbec6488a610

fixes review commit hash - b310ae785521695bbc6ea952655e28cd0eb5a046

Scope

The following smart contracts were in scope of the audit:

- `EscrowAssetStorage`
- `HyperVaultRouter`
- `HyperliquidEscrow`
- `L1EscrowActions`
- `WrappedVaultShare`

7. Executive Summary

Over the course of the security review, unforgiven, zark, aslanbek engaged with Blueberry to review Blueberry. In this period of time a total of **9** issues were uncovered.

Protocol Summary

Protocol Name	Blueberry
Repository	https://github.com/Blueberryfi/blueberry-v2-contracts
Date	May 16th 2025 - May 19th 2025
Protocol Type	Lending

Findings Count

Severity	**Amount**
High	1
Low	8
Total Findings	**9**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Escrow.tvl() does not add in-flight USDC amount	High	Resolved
[L-01]	Removing unsupported asset might wrongly remove USDC	Low	Resolved
[L-02]	minDepositValue uses inadequate bit size for large deposits	Low	Resolved
[L-03]	maxRedeemable uses incorrect rounding method	Low	Resolved
[L-04]	maxRedeemable underestimates shares due to uncollected fees	Low	Resolved
[L-05]	Fee calculation includes historical balances of new assets	Low	Resolved
[L-06]	Slippage protection may reference wrong asset after withdrawal changes	Low	Acknowledged
[L-07]	Dead code in bridgeToL1	Low	Resolved
[L-08]	Asset removal ignores in-flight bridge amounts	Low	Resolved

8. Findings

8.1. High Findings

[H-01] `Escrow.tvl()` does not add in-flight USDC amount

Severity

Impact: Medium

Likelihood: High

Description

The `tvl()` function in `HyperliquidEscrow` contract does not include the USDC amount that is currently being bridged (in-flight) when calculating the total value locked (TVL). For other assets, if a bridge is happening in the same block, that amount is added to the TVL. But for USDC, this step is skipped. This means the reported TVL is lower than it should be when USDC is being bridged.

```
function tvl() external view override returns (uint256 tvl_) {
            --Snipped--
            if (assetIndex == USDC_SPOT_INDEX) {
                tvl_ += IERC20(assetAddr).balanceOf(address(this)) * evmScaling;
            } else {
                uint256 rate = getRate(details.spotMarket, details.szDecimals);
                uint256 balance = IERC20(assetAddr).balanceOf(address
                  (this)) * evmScaling;


                                uint256 lastBridgeToL1Block = $$.inFlightBridge[asset
                // If we are still in the L1 bridge period
                //(same EVM block as last bridge action took place), we need to add th
                if (block.number == lastBridgeToL1Block) {

                                    balance += $$.inFlightBridge[assetIndex].amou
                }

                balance += _spotAssetBalance(uint64(assetIndex));
                tvl_ += balance.mulWadDown(rate);
            }
        }
    }
```

As a result, any user who deposits or redeems in the same block when a USDC bridge is happening will see an incorrect TVL and share price. This can cause users to lose funds or unfairly receive more than they should.

POC 1

This is a test unit regarding this issue. File should be in `test/hyperliquid` folder. test_tvl.txt

POC 2

You can find the PoC here and run `forge test --mt testTVLIgnoresUSDCBridged`.

Recommendations

Fix the logic so that the in-flight USDC amount is also added to the TVL if it's being bridged in the current block, just like other assets. This will make the TVL calculation correct and consistent.

8.2. Low Findings

[L-01] Removing unsupported asset might wrongly remove USDC

The `removeAsset()` function uses `$.assetIndexes[asset]` to get the asset index. If the asset address is not actually supported, the mapping returns `0` by default — which is the index for USDC. This can result in mistakenly removing USDC when an unsupported or incorrect address is passed to the function.

```
function removeAsset(address asset) external onlyOwner {
        V1Storage storage $ = _getV1Storage();
        uint64 assetIndex_ = $.assetIndexes[asset];

        // Withdraw asset cannot be set to address(0) once it is set
        require(asset != $.withdrawAsset, Errors.INVALID_OPERATION());

        if (assetIndex_ == USDC_EVM_SPOT_INDEX) $.usdcSupported = false;
        delete $.assetIndexes[asset];
        delete $.assetDetails[assetIndex_];
        $.supportedAssets.remove(assetIndex_);

        uint256 len = $.escrows.length;
        for (uint256 i = 0; i < len; ++i) {
            HyperliquidEscrow($.escrows[i]).removeAsset(assetIndex_);
        }

        emit AssetRemoved(assetIndex_);
    }
```

This same pattern is also present in the `setWithdrawAsset()` function. It fetches the asset index using `$.assetIndexes[asset]` without first verifying that the asset exists. If an unsupported asset is provided, the function will treat it as index 0, and mistakenly allow USDC to be set as the withdrawal asset again.

Recommendations: Add a validation step to ensure the asset is actually supported before using the index returned by the mapping. Specifically:

- Check that `$.assetDetails[assetIndex_].evmContract == asset` to confirm the asset is registered.
- Reject the operation if the asset is not properly mapped.

This pattern should be fixed in **both** `removeAsset()` and `setWithdrawAsset()` to prevent unintended behavior and accidental removal or misuse of USDC.

[L-02] `minDepositValue` uses inadequate bit size for large deposits

The `minDepositValue` field in the `V1Storage` struct is defined as `uint64`, which has a maximum value of approximately `18.44e18`. This may be insufficient for routers that need to handle large deposit amounts at a minimum, meaning more than 20e18 USD. In any case, such a limitation of the variable can be a problem.

```
struct V1Storage {
        /// @notice The last time the fees were accrued
        uint64 lastFeeCollectionTimestamp;
        /// @notice The management fee in basis points
        uint64 managementFeeBps;
        /// @notice The minimum value in USD that can be deposited into the
        // vault scaled to 1e18
@>        uint64 minDepositValue;
```

Consider using a larger `uint` type for the `minDepositValue`.

[L-03] `maxRedeemable` uses incorrect rounding method

The `maxRedeemable` function in `HyperVaultRouter.sol` uses `mulDivDown` for calculating the maximum redeemable shares, which rounds down the result. However, in this context, rounding up would be more appropriate since it would allow users to redeem more shares for the same amount of tokens.

```
function maxRedeemable() external view override returns (uint256) {
        // ...

        // Calculate the max redeemable amount based on the USD value of the
        // withdraw asset
        uint256 usdValue = maxWithdraw.mulWadDown(rate * scaler);
@>        return usdValue.mulDivDown(_shareSupply(), tvl());
    }
```

Consider rounding up so more shares to be redeemed for the same amount of assets, in favour of the protocol.

[L-04] `maxRedeemable` underestimates shares due to uncollected fees

The `maxRedeemable` function in `HyperVaultRouter.sol` calculates the maximum number of shares that can be redeemed based on the current TVL and share supply, but fails to account for uncollected management fees. This leads to an underestimation of the actual redeemable shares.

```
function maxRedeemable() external view override returns (uint256) {

        // Calculate the max redeemable amount based on the USD value of the
        // withdraw asset
        uint256 usdValue = maxWithdraw.mulWadDown(rate * scaler);
@>        return usdValue.mulDivDown(_shareSupply(), tvl
//()); // # this `_shareSupply()` is outdated
    }
```

In order to fix this issue, consider calling `_takeFee()` inside `maxRedeemable`().

[L-05] Fee calculation includes historical balances of new assets

The `addAsset` function in `HyperVaultRouter.sol` has an issue with fee calculation when adding new assets in a router that already has balances in the escrow will in result overcharging fees for this new asset for the already. The problem occurs because:

1. When a new asset is added, its balance is immediately included in TVL calculations.
2. The fee calculation uses lastFeeCollectionTimestamp to determine the time period for fee accrual.
3. If an escrow already holds balance of an unsupported token that is later added, the protocol will calculate fees as if that the balance existed since `lastFeeCollectionTimestamp`, not from when the asset was actually added.

```
function addAsset(
      addressassetAddr,
      uint32assetIndex_,
      uint32spotMarket
    ) external onlyOwner {
        // ...

        // Iterate through all the escrows to add supported assets
        uint256 len = $.escrows.length;
        for (uint256 i = 0; i < len; ++i) {
            HyperliquidEscrow($.escrows[i]).addAsset(assetIndex_, details);
        }

        emit AssetAdded(assetIndex_, details);
    }
```

To fix this issue, consider calling `_takeFees()` inside `addAsset()` function.

[L-06] Slippage protection may reference wrong asset after withdrawal changes

The `redeem` function in `HyperVaultRouter.sol` has an issue with slippage protection when the withdraw asset is changed between the time a user submits their transaction and when it is executed. This occurs because:

1. The user is submitting his tx for a redemption, seeing that the `withdrawAsset` is `X`.
2. However, the change of `withdrawAsset` is happening before the tx is executed.
3. So, now `redeem` is being passed with `minOut` reffering to other tokens' unit.

```
function redeem(
      uint256shares,
      uint256minOut
    ) external override nonReentrant returns (uint256 amount
        // ...

        // Convert the USD amount to withdraw to the withdraw asset amount
        HyperliquidEscrow escrow = HyperliquidEscrow
          ($.escrows[depositEscrowIndex()]);
        uint256 scaler = 10 ** (18 - details.evmDecimals);
        uint256 rate = _getRate(address(escrow), assetIndex_, details);
        amount = usdAmount.divWadDown(rate * scaler);
        require(amount > 0, Errors.AMOUNT_ZERO());
        require(amount >= minOut, Errors.SLIPPAGE_TOO_HIGH());

        // ...
    }
```

The impact of this is that for example someone will decide that he wants to slippage to `10e6` USDC, but if the `withdrawAsset` change, the transaction will be executed and the slippage will be `10e6` PURR.

To fix this issue, consider taking as a parameter the `withdrawAsset` in `redeem`.

[L-07] Dead code in `bridgeToL1`

`bridgeToL1` tracks bridged assets:

```
uint256 evmBlock = block.number;
        uint64 lastBridgeToL1Block = $$.inFlightBridge[assetIndex].blockNumber;


        if (evmBlock != lastBridgeToL1Block) {
            $$.inFlightBridge[assetIndex] = InflightBridge({blockNumber: uint64
              (evmBlock), amount: amountAdjusted});
        } else {
            $$.inFlightBridge[assetIndex].amount += amountAdjusted;
        }
```

The last line can be reached only if `bridgeToL1` is called twice within the same EVM block, which is impossible, because the function has `singleActionBlock` modifier.

[L-08] Asset removal ignores in-flight bridge amounts

The `removeAsset` function in `HyperVaultRouter.sol` fails to account for assets that are currently being bridged from HyperEVM L2 to HyperCore L1. This is problematic because:

1. The protocol tracks in-flight bridge amounts in the V1L1EscrowActionsStorage struct:

```
struct InflightBridge {
        /// @notice The evm block number that the asset was sent to L1
        uint64 blockNumber;
        /// @notice The amount of the asset that is in-flight to L1
        uint256 amount;
    }
```

2. But upon removal, it is checked only if the balance in the escrow or the spot balance is 0, and spot balance is not taking into account the in flight amounts that will be in spot balance in the next block.

```
function _spotAssetBalance(uint64 token) internal view override returns
      (uint256) {
        V1AssetStorage storage $ = _getV1AssetStorage();
        (bool success, bytes memory result) =
            SPOT_BALANCE_PRECOMPILE_ADDRESS.staticcall(abi.encode(address
              (this), token));
        require(success, Errors.PRECOMPILE_CALL_FAILED());
        SpotBalance memory balance = abi.decode(result, (SpotBalance));

        if (token == USDC_SPOT_INDEX) {
            return balance.total * USDC_SPOT_SCALING;
        }

        uint256 scaler = 10 ** (18 - $.assetDetails[token].weiDecimals);
        return balance.total * scaler;
    }
```

In this way, it is possible that `LIQUIDITY_ADMIN` has made a bridge of the full amount of Escrow (so all balances will be zero, except the inflight struct) and at that point, if `removeAsset` is triggered, it will not revert and the assets will be locked.

See the PoC here and run `forge test --mt testRemoveAssetDoesntSeeSpot`.

Recommendations: Consider taking into account and checking that in flight amounts are also zero.