Blueberry Security Review

Pashov Audit Group

Conducted by: Mario Poneder, btk, Ch_301, t.aksoy, sl1, 0xBugSlayer, ayeslick

April 30th 2025 - May 4th 2025

Contents

1. About Pashov Audit Group 3
2. Disclaimer 3
3. Introduction 3
4. About Blueberry 3
5. Risk Classification 4
   5.1. Impact 4
   5.2. Likelihood 4
   5.3. Action required for severity levels 5
6. Security Assessment Summary 5
7. Executive Summary 6
8. Findings 9
   8.1. Critical Findings 9
      [C-01] Incorrect conversion from USD to asset during redemption 9
      [C-02] Missing asset index check allows any token to mint shares 10
   8.2. High Findings 12
      [H-01] getRate() ignores szDecimals 12
      [H-02] Missing custom logic for USDC spot market 13
      [H-03] Users can exploit TVL during EVM to L1 transfer causing share inflation 14
      [H-04] Stale inFlightBridgeAmounts cause incorrect tvl calculations 16
   8.3. Medium Findings 18
      [M-01] _validateSpotMarket incorrectly validates spot market 18
      [M-02] Small Amounts Burned When Bridging to L1 19
      [M-03] Missing slippage protection in deposit/redeem functions 20
      [M-04] maxRedeemable calculation misses withdrawAsset rate 20
   8.4. Low Findings 22

[L-01] Unsupported asset can be set in withdraw asset by index collision 22
[L-02] Risk of withdrawAsset being set to address(0) post asset removal 22
[L-03] Redundant condition check 23
[L-04] upgrade method not available in OpenZeppelin ProxyAdmin contract 23
[L-05] Deployment script lacks deployer == OWNER consistency check 24
[L-06] BURNER_ROLE not assigned in deployment script 24
[L-07] LIQUIDITY_ADMIN_ROLE not assigned during escrow initialization 25
[L-08] WrappedVaultShare constructor misassigns ROUTER 25

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 - bcb1a056864226e433705f8e25926a126d21d38b

fixes review commit hash - d299166b968039bec4978fdd8a4c782d113dae9b

Scope

The following smart contracts were in scope of the audit:

- `DeployRouterTestnet`
- `UpgradeEscrows`
- `UpgradeRouter`
- `EscrowAssetStorage`
- `HyperVaultRouter`
- `HyperliquidEscrow`
- `L1EscrowActions`
- `WrappedVaultShare`

7. Executive Summary

Over the course of the security review, Mario Poneder, btk, Ch_301, t.aksoy, sl1, 0xBugSlayer, ayeslick engaged with Blueberry to review Blueberry. In this period of time a total of **18** issues were uncovered.

Protocol Summary

Protocol Name	Blueberry
Repository	https://github.com/Blueberryfi/blueberry-v2-contracts
Date	April 30th 2025 - May 4th 2025
Protocol Type	Lending

Findings Count

Severity	**Amount**
Critical	2
High	4
Medium	4
Low	8
Total Findings	**18**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Incorrect conversion from USD to asset during redemption	Critical	Resolved
[C-02]	Missing asset index check allows any token to mint shares	Critical	Resolved
[H-01]	getRate() ignores szDecimals	High	Resolved
[H-02]	Missing custom logic for USDC spot market	High	Resolved
[H-03]	Users can exploit TVL during EVM to L1 transfer causing share inflation	High	Resolved
[H-04]	Stale inFlightBridgeAmounts cause incorrect tvl calculations	High	Resolved
[M-01]	_validateSpotMarket incorrectly validates spot market	Medium	Resolved
[M-02]	Small Amounts Burned When Bridging to L1	Medium	Resolved
[M-03]	Missing slippage protection in deposit/redeem functions	Medium	Resolved
[M-04]	maxRedeemable calculation misses withdrawAsset rate	Medium	Resolved
[L-01]	Unsupported asset can be set in withdraw asset by index collision	Low	Resolved
[L-02]	Risk of withdrawAsset being set to address(0) post asset removal	Low	Resolved
[L-03]	Redundant condition check	Low	Resolved
[L-04]	upgrade method not available in	Low	Resolved

	OpenZeppelin ProxyAdmin contract		
[L-05]	Deployment script lacks deployer == OWNER consistency check	Low	Resolved
[L-06]	BURNER_ROLE not assigned in deployment script	Low	Resolved
[L-07]	LIQUIDITY_ADMIN_ROLE not assigned during escrow initialization	Low	Resolved
[L-08]	WrappedVaultShare constructor misassigns ROUTER	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Incorrect conversion from USD to asset during redemption

Severity

Impact: High

Likelihood: High

Description

In the deposit() function, the conversion from asset amount to USD value is performed by multiplying the spot rate: `uint256 usdValue = escrow.getRate(details.spotMarket).mulWadDown(amount * scaler);` in the redeem() function, the conversion from USD back to asset is incorrectly also performed via multiplication: `amount = escrow.getRate(details.spotMarket).mulWadDown(usdAmount);`

This causes the user to receive more tokens than they originally deposited, leading to value inflation. For example, if the rate is 2 * 1e18 (i.e., 1 asset = 2 USD), and a user deposits 1e18 worth of assets (USD value = 2e18), the redeem logic will return: `amount = 2e18 * 2e18 / 1e18 = 4e18`

This results in a 4x increase in asset amount, instead of the expected 1x. This error can cause severe financial loss to the protocol as users can exploit it to mint more tokens than they deposit.

Recommendations

Replace the multiplication in the redeem() function with division to correctly convert from USD back to asset:

```
amount = usdAmount.divWadDown(escrow.getRate(details.spotMarket));
```

[C-02] Missing asset index check allows any token to mint shares

Severity

Impact: High

Likelihood: High

Description

The deposit function in the HyperVaultRouter contract performs a check to ensure that the deposited token is supported via the _isAssetSupported method, using the asset index retrieved from $.assetIndexes[asset]. However, this check is flawed due to the following logic:

```
uint64 assetIndex_ = $.assetIndexes[asset];
        require(_isAssetSupported
          ($, assetIndex_), Errors.COLLATERAL_NOT_SUPPORTED());
```

If a token is not registered (i.e., not added via addAsset), the mapping $.assetIndexes[asset] returns the default value of 0. Since USDC is hardcoded to index 0 (USDC_SPOT_INDEX = 0), and USDC is a valid and supported asset, this check will falsely pass for any unsupported ERC20 token whose address is not in the assetIndexes mapping.

As a result:

- Any user can call deposit() with a random, potentially worthless ERC20 token.
- The protocol will incorrectly treat the token as USDC.
- Shares will be minted based on the USDC value.

POC: https://gist.github.com/tevrat-aksoy/da734f144cd0176c41a3161ca74ab791

```
function testAnyTokenCanDeposited() public {
        //  First add the asset to the router
        router.addAsset(address(asset), 0, 0);
        router.setWithdrawAsset(address(asset));

        //  Then mint tokens to Alice
        asset.mint(alice, 100e8);

        vm.startPrank(alice);
        asset.approve(address(router), 100e8);
        router.deposit(usdcADDRESS, 100e8);

        assertEq(shareToken.balanceOf(address
          (alice)), 100e18, "alice should have 100e8 shares");

        assertEq(router.tvl(), 100e18, "alice should have 100e8 shares");

        // Create random token and deposit
        MockERC20 randomERC = new MockERC20("Random", "RND", 8);
        randomERC.mint(bob, 100e8);
        vm.startPrank(bob);

        randomERC.approve(address(router), 100e8);
        router.deposit(address(randomERC), 100e8);

        assertEq(shareToken.balanceOf(address
          (bob)), 100e18, "bob should have 100e8 shares");
        // TVL didnt change but share token supply increased
        assertEq(router.tvl(), 100e18, "alice should have 100e8 shares");
    }
```

Recommendations

Disallow assetIndex_ == 0 unless asset is known to be USDC: Add a special-case check to validate the asset is USDC when index is 0.

8.2. High Findings

[H-01] `getRate()` ignores `szDecimals`

Severity

Impact: High

Likelihood: Medium

Description

The getRate() function retrieves a spot price from a precompile and scales it assuming a fixed USDC_SPOT_DECIMALS = 8:

```
uint8 public constant USDC_SPOT_DECIMALS = 8;
uint256 public constant USDC_SPOT_SCALING = 10 ** (18 - USDC_SPOT_DECIMALS);

function getRate(uint32 spotMarket) public view override returns (uint256) {
    (bool success, bytes memory result) = SPOT_PX_PRECOMPILE_ADDRESS.staticcall
      (abi.encode(spotMarket));
    require(success, Errors.PRECOMPILE_CALL_FAILED());
    uint256 scaledRate = uint256(abi.decode(result,
      (uint64))) * USDC_SPOT_SCALING;
    return scaledRate;
}
```

However, in the Hyperliquid protocol, the raw spot price returned by the precompile does not always conform to 8 decimals. Instead, the number of meaningful price decimals is dynamically constrained by the szDecimals value of the asset involved:

For a spot token with szDecimals = 0, prices can go up to 8 decimals (e.g., 0.108 → 10845000)

For szDecimals = 2, the price can have at most 6 decimals (i.e., 8 - szDecimals)

Example: Token: HFUN (szDecimals = 2) Price: $37.07 Precompile returns: 37073000 Interpreted as 0.37073000 if assuming 8 decimals — which is incorrect Should actually be treated as 37.073000 with only 6 decimals

`cast call 0x0000000000000000000000000000000000000808 0x0000000000000000000000000000000000000000000000000000000000000001 --rpc-url https://rpc.hyperliquid.xyz/evm` --> RESULT: 37073000

Recommendations

szDecimal should be applied to the precompile result:

```
function getRate(
      uint32spotMarket,
      uint64assetIndex
    ) public view override returns (uint256

AssetDetails memory details = $.assetDetails[assetIndex];

        (
          boolsuccess,
          bytesmemoryresult
        ) = SPOT_PX_PRECOMPILE_ADDRESS.staticcall(abi.encode(spotMarket
        require(success, Errors.PRECOMPILE_CALL_FAILED());
        uint256 scaledRate = uint256(abi.decode(result,
          (uint64))) * USDC_SPOT_SCALING* 10 ** (details.szDecimals) ;
        return scaledRate;

}
```

[H-02] Missing custom logic for USDC spot market

Severity

Impact: Medium

Likelihood: High

Description

When adding a supported asset using addAsset(), the spotMarket ID must be provided. This market ID is later used in getRate() to fetch the USD value of the deposited asset through a precompile call. However, in the case of USDC, there is no valid spot market to get its price:

```
Universe: [
  {'tokens': [1, 0], 'name': 'PURR/USDC', 'index': 0, 'isCanonical': True},
  {'tokens': [2, 0], 'name': '@1', 'index': 1, 'isCanonical': False},
  {'tokens': [3, 0], 'name': '@2', 'index': 2, 'isCanonical': False}
]
```

All listed tokens are traded against USDC, implying that USDC is the quote asset. Therefore, it cannot have a valid market where its price is retrieved in USD, since it's already denominated in USD.

Recommendations

Implement a special case in the getRate() function (or upstream during asset addition) to return a constant value of 1e18 for USDC, representing a 1:1 USD value.

[H-03] Users can exploit TVL during EVM to L1 transfer causing share inflation

Severity

Impact: High

Likelihood: Medium

Description

Users can exploit TVL calculation during EVM to L1 transfer, leading to share inflation

The `HyperliquidEscrow` contract is responsible for holding assets and calculating the total value locked (TVL) for a vault that interacts with both EVM and L1 (core) systems.

The TVL is used as the basis for share issuance and redemption, and is calculated in the `tvl()` function by summing balances from both EVM and L1 sources, including spot and perpetual balances.

A critical aspect of this design is the transfer of assets from the EVM to the L1 system. According to The Not-So-Definitive guide to Hyperliquid Precompiles:

> when tokens are transferred from the EVM to L1, there is a "pre-crediting" period where the tokens have left the EVM but have not yet been credited to the L1 spot balance.

During this window, the TVL calculation in `tvl()` will underreport the true value of assets under management, as the tokens are not reflected in either the EVM or L1 balances. This creates a window of opportunity for users to exploit the temporarily reduced TVL.

```
function tvl() external view override returns (uint256 tvl_) {
        // Get the equity, spot, and perp USD balances
        tvl_ = vaultEquity(); // scaled to 1e18
        tvl_ += usdSpotBalance(); // scaled to 1e18
        tvl_ += usdPerpsBalance(); // scaled to 1e18

        V1AssetStorage storage $ = _getV1AssetStorage();

        // Iterate through all supported assets calculate their contract and
        // spot value
        //     and add them to the tvl
        for (uint256 i = 0; i < $.supportedAssets.length(); i++) {
            uint64 assetIndex = uint64($.supportedAssets.at(i));
            AssetDetails memory details = $.assetDetails[assetIndex];
            address assetAddr = details.evmContract;

            uint256 evmScaling = 10 ** (18 - details.evmDecimals);

            if (assetIndex == USDC_SPOT_INDEX) {
                // If the asset is USDC we only need to get the contract balance
                // since we already queried the spot balance
                tvl_ += IERC20(assetAddr).balanceOf(address(this)) * evmScaling;
            } else {
                uint256 rate = getRate(details.spotMarket);
                uint256 balance = IERC20(assetAddr).balanceOf(address
                  (this)) * evmScaling;
                balance += _spotAssetBalance(uint64(assetIndex));
                tvl_ += balance.mulWadDown(rate);
            }
        }
    }
```

Proof of Concept

1. The system has 1000 USDC and 1000 shares outstanding.
2. The `LIQUIDITY_ADMIN_ROLE` initiates a transfer of 500 USDC from the EVM to the L1 system.
    - The EVM balance decreases by 500 USDC immediately.
    - The L1 spot balance has not yet been credited (pre-crediting state).
    - The `tvl()` function now reports only 500 USDC.
3. A user deposits 100 USDC during this pre-crediting window.
    - The share price is calculated as 500 USDC / 1000 shares = 0.5 USDC/share.
    - The user receives 200 shares for their 100 USDC deposit.
4. After a few blocks, the L1 credits the 500 USDC.

- The true TVL is now 1100 USDC (1000 original + 100 deposit).
- There are 1200 shares outstanding.
- The user can redeem their 200 shares for 183 USDC (200/1200 * 1100), realizing a profit of 83 USDC.

5. The protocol and other users are diluted, and the attacker profits at their expense.

Recommendations

To prevent this exploit, there are two main approaches:

1. **Track pre-credited balances internally:** Maintain an internal record of assets in transit and include them in the TVL calculation until they are credited on L1.
2. **Pause deposits/redemptions during pre-crediting:** Temporarily disable deposit and redemption functions for a few blocks after initiating an EVM to L1 transfer, until the crediting is confirmed.

[H-04] Stale `inFlightBridgeAmounts` cause incorrect `tvl` calculations

Severity

Impact: High

Likelihood: Medium

Description

The `inFlightBridgeAmounts[assetIndex]` mapping in the `L1EscrowActions` contract is not cleared after assets are settled on L1. This leads to outdated and unrelated in-flight amounts lingering in the mapping, which are then incorrectly included in `tvl` calculations.

Steps to reproduce

1. **Block 1**: Execute `bridgeToL1` for `assetA`. The `inFlightBridgeAmounts[assetA]` mapping is updated with the corresponding bridged amount.

2. **Block 2**: Asset A settles on L1. However, the `inFlightBridgeAmounts[assetA]` mapping remains uncleared.
3. **Block 3**: Execute `bridgeToL1` for `assetB`. The `inFlightBridgeAmounts[assetB]` mapping is updated with the new bridged amount.
4. **Block 3**: Compute `tvl`. The calculation incorrectly factors in the stale `inFlightBridgeAmounts[assetA]` alongside the active `inFlightBridgeAmounts[assetB]`.

This behavior results in inflated `tvl` reporting which has secondary effects on the protocol's share accounting.

Recommendations

To resolve this issue, consider implementing a 2D mapping structure to track in-flight amounts by both asset index and block number:

```
mapping(uint64 assetIndex => mapping
  (uint256 blockNumber => uint256 amount)) inFlightBridgeAmounts;
```

8.3. Medium Findings

[M-01] `_validateSpotMarket` incorrectly validates spot market

Severity

Impact: Medium

Likelihood: Medium

Description

The *validateSpotMarket() function is intended to ensure that the given spotMarket is valid for the provided assetIndex*, by checking if the market contains both the asset and USDC (as the system assumes all assets are priced against USDC). However, the current logic uses an OR condition:

```
if
  (spotInfo.tokens[i] == USDC_SPOT_INDEX || spotInfo.tokens[i] == assetIndex_) {
    return true;
}
```

Since all valid spot markets pair assets against USDC, USDC_SPOT_INDEX (0) will always appear in one of the tokens. This causes the function to return true even if the market is unrelated to the given asset.

Example:

assetIndex_ = 5 (e.g., some token)

spotInfo.tokens = [2, 0] (a market for token 2 against USDC)

The check will return true because tokens[1] == USDC_SPOT_INDEX, even though 5 is not involved in the market at all.

Recommendations

Change the validation logic to ensure that one of the tokens is USDC_SPOT_INDEX and the other is assetIndex_:

```
function _validateSpotMarket
  (uint64 assetIndex_, uint32 spotMarket) internal view returns (bool) {
    (
      boolsuccess,
      bytesmemoryresult
    ) = SPOT_INFO_PRECOMPILE_ADDRESS.staticcall(abi.encode(spotMarket
    require(success, Errors.PRECOMPILE_CALL_FAILED());
    SpotInfo memory spotInfo = abi.decode(result, (SpotInfo));

    return (

          (spotInfo.tokens[0] == assetIndex_ && spotInfo.tokens[1] == USDC_SPOT_INDEX)

          (spotInfo.tokens[1] == assetIndex_ && spotInfo.tokens[0] == USDC_SPOT_INDEX)
    );
}
```

[M-02] Small Amounts Burned When Bridging to L1

Severity

Impact: Medium

Likelihood: Medium

Description

The bridgeToL1() function transfers tokens from the escrow to a system address on the EVM to initiate a bridge to L1. However, if the EVM-side token has extra decimal precision (i.e., more than the expected extraEvmWeiDecimals), non-round values (not divisible cleanly by 10**extraEvmWeiDecimals) will result in burned tokens. This behavior is noted in the protocol's caveats and is systemic to how logs are parsed and how L1 crediting works. However, the bridge logic does not pre-adjust or sanitize these amounts. This leads to permanent token loss.

Recommendations

Before sending, round down the amount like this:

```
uint256 factor = 10 ** details.extraEvmWeiDecimals;
amounts[i] = amounts[i] - (amounts[i] % factor); IERC20
  (details.evmContract).transfer(_assetSystemAddr(assetIndexes[i]), amounts[i]);
```

[M-03] Missing slippage protection in deposit/redeem functions

Severity

Impact: High

Likelihood: Low

Description

The `HyperVaultRouter` contract implements a vault system where users can deposit various assets and receive share tokens in return. The share calculation in the `deposit()` function relies on the USD value of deposited assets, which is determined by querying rates from the escrow contract. However, the function lacks slippage protection, making users vulnerable to sandwich attacks and unfavorable rate manipulation.

```
function deposit(
      addressasset,
      uint256amount
    ) external override nonReentrant returns (uint256 shares
    function redeem(uint256 shares) external override nonReentrant returns
      (uint256 amount) {
```

Since there is no minimum output parameter for shares, users have no control over the minimum number of shares they will receive. This is particularly problematic because:

1. The rate can fluctuate between transaction submission and execution.
2. MEV bots can sandwich the transaction to manipulate the rate.

Recommendations

Add a `minOut` parameter to the `deposit()` and `redeem()` functions to allow users to specify the minimum number of shares/assets they expect to receive:

[M-04] `maxRedeemable` calculation misses `withdrawAsset` rate

Severity

Impact: Low

Likelihood: High

Description

The `maxRedeemable` function calculates the maximum redeemable amount using the formula:

```
return maxWithdraw.mulDivDown(_shareSupply(), tvl());
```

This calculation assumes that the `withdrawAsset` value is directly proportional to the TVL. However, it does not account for the rate of the `withdrawAsset` (e.g., its USD value) and its decimals. If the `withdrawAsset` has a different rate, the calculation will result in incorrect values, leading to potential misaccounting and integration issues for anyone relying on `maxRedeemable`.

Recommendations

It is recommended to factor in the `withdrawAsset` rate (see `getRate`) and correct decimal scaling when calculating the redeemable amount.

8.4. Low Findings

[L-01] Unsupported asset can be set in withdraw asset by index collision

The setWithdrawAsset() function allows setting any ERC20 token as the withdrawable asset, but it checks support using the index returned by $.assetIndexes[asset]. If the given asset is not supported, assetIndexes[asset] will return the default value 0, which corresponds to the USDC index. This causes _isAssetSupported() to incorrectly return true, allowing an unsupported token to be set. As a result, the redeem() function will fail when trying to compute rates or transfer the redeemable asset, resulting in withdrawals.

```
function setWithdrawAsset(address asset) external onlyOwner {
        V1Storage storage $ = _getV1Storage();
        uint64 assetIndex_ = $.assetIndexes[asset];
        require(_isAssetSupported
          ($, assetIndex_), Errors.COLLATERAL_NOT_SUPPORTED());
```

Add a check that if assetIndex_ is 0 asset address should be USDC.

[L-02] Risk of `withdrawAsset` being set to `address(0)` post asset removal

The `removeAsset()` function in the `HyperVaultRouter` contract sets the `withdrawAsset` to `address(0)` if the asset being removed is the current `withdrawAsset`:

```
if (asset == $.withdrawAsset) $.withdrawAsset = address(0);
```

This creates a risk where the owner could call `removeAsset` to remove the `withdrawAsset` and subsequently invoke the `renounceOwnership()` function. If ownership is renounced without assigning a new `withdrawAsset`, the `redeem()` function will become permanently unusable due to the following check:

```
require($.withdrawAsset != address(0), Errors.ADDRESS_ZERO());
```

This can be resolved by: Prevent renouncing ownership with `withdrawAsset` set to `address(0)`, or set a new withdrawAsset before removing it.

[L-03] Redundant condition check

The `_scaleToSpotDecimals()` function in `HyperliquidEscrow` contains a redundant condition check for decimal scaling between perpetual and spot markets. The function checks if `USDC_PERP_DECIMALS > USDC_SPOT_DECIMALS`, but this condition will never be true since `USDC_PERP_DECIMALS` is defined as 6 and `USDC_SPOT_DECIMALS` is defined as 8 in the contract constants:

```
function _scaleToSpotDecimals(uint64 amount_) internal pure returns
      (uint64) {
        return (USDC_PERP_DECIMALS > USDC_SPOT_DECIMALS)
            ? uint64(amount_ / (10 **
              (USDC_PERP_DECIMALS - USDC_SPOT_DECIMALS)))
            : uint64(amount_ * (10 **
              (USDC_SPOT_DECIMALS - USDC_PERP_DECIMALS)));
    }
```

Recommendation: Simplify the function by removing the conditional logic and directly implementing the scaling operation:

```
function _scaleToSpotDecimals(uint64 amount_) internal pure returns (uint64) {
    return uint64(amount_ * (10 ** (USDC_SPOT_DECIMALS - USDC_PERP_DECIMALS)));
}
```

[L-04] `upgrade` method not available in OpenZeppelin `ProxyAdmin` contract

The `UpgradeRouterScript` script attempts to call the `upgrade` method on the `ProxyAdmin` contract:

```
proxyAdmin.upgrade(ROUTER_PROXY_ADDRESS, address(router));
```

However, the utilized OpenZeppelin `ProxyAdmin` contract (v5.2.0) does not include an `upgrade` method. Instead, it provides the `upgradeAndCall` method,

which requires an additional `bytes memory data` parameter.

It is recommended to replace the `upgrade` method call with `upgradeAndCall` and pass an empty `data` parameter.

[L-05] Deployment script lacks `deployer == OWNER` consistency check

The deployment script (`DeployRouterTestnet.s.sol`) uses `deployer` and `OWNER` interchangeably as the owner and proxy admin during contract deployments. However, it does not verify that `deployer == OWNER`. This inconsistency can lead to permission issues, particularly during the execution of router and escrow upgrade scripts, as these rely on the correct owner and proxy admin permissions.

It is recommended to add a check to ensure `deployer == OWNER` at the start of the script:

```
require(deployer == OWNER, "Deployer must match OWNER");
```

[L-06] `BURNER_ROLE` not assigned in deployment script

The deployment script (`DeployRouterTestnet.s.sol`) sets the `MINTER_ROLE` for the `HyperVaultRouter` on the `WrappedVaultShare` token but neglects to assign the `BURNER_ROLE`. Without the `BURNER_ROLE`, the router cannot burn share tokens, which is critical for proper functionality during withdrawals.

It is recommended to add the following line to the deployment script to assign the `BURNER_ROLE` to the router:

```
shareToken.grantRole(shareToken.BURNER_ROLE(), address(router));
```

[L-07] `LIQUIDITY_ADMIN_ROLE` not assigned during escrow initialization

The `LIQUIDITY_ADMIN_ROLE`, required for critical liquidity management functions (e.g., `bridgeToL1`, `trade`), is never granted during the initialization of the escrow contract. This leaves no account authorized to perform these actions, rendering the contract's liquidity operations unusable.

It is recommended to grant the `LIQUIDITY_ADMIN_ROLE` to an admin address during contract initialization or in the deployment script.

[L-08] `WrappedVaultShare` constructor misassigns ROUTER

`WrappedVaultShare` misassigns the constructor argument to itself instead of the immutable state variable,

```
constructor
   (string memory name, string memory symbol, address router, address admin)
        MintableToken(name, symbol, 18, admin)
    {
        router = ROUTER;
    }
```

As a result, the code fails to compile.

Recommendation:

change to: `ROUTER = router;`