Blueberry Security Review

Pashov Audit Group

Conducted by: unforgiven, ZanyBonzy, 0x37, crunter, PlamenTSV, km

March 12th 2025 - March 15th 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 - 8947ad027c5456fb968d30e0148c773402d43458

fixes review commit hash - 03dc79ddbacac2a4a1b194a06a7f9dfca6ad2fcb

Scope

The following smart contracts were in scope of the audit:

- `HyperEvmVault`
- `VaultEscrow`

7. Executive Summary

Over the course of the security review, unforgiven, ZanyBonzy, 0x37, crunter, PlamenTSV, km engaged with Blueberry to review Blueberry. In this period of time a total of **14** issues were uncovered.

Protocol Summary

Protocol Name	Blueberry
Repository	https://github.com/Blueberryfi/blueberry-v2-contracts
Date	March 12th 2025 - March 15th 2025
Protocol Type	Lending

Findings Count

Severity	**Amount**
High	5
Medium	4
Low	5
Total Findings	**14**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Unapproved requests are not deducted from total assets	High	Resolved
[H-02]	Withdraw check can be bypassed	High	Resolved
[H-03]	Incorrect erc20 transfer implementation between evm and core chains	High	Resolved
[H-04]	Malicious users may block other withdrawals	High	Resolved
[H-05]	User can drain escrow via approved withdrawals	High	Resolved
[M-01]	setManagementFeeBps() fails to update fee state	Medium	Resolved
[M-02]	Incorrect rounding in mint()	Medium	Resolved
[M-03]	Wrong fee share calculations in _takeFee()	Medium	Resolved
[M-04]	Protocol fee rounding can be weaponized for minimal earnings	Medium	Resolved
[L-01]	Add checks for zero returned shares	Low	Resolved
[L-02]	maxwithdraw() ignores pending redemptions	Low	Resolved
[L-03]	Preview functions fail with supply=0	Low	Resolved
[L-04]	Protocol may charge more fees than expected	Low	Resolved
[L-05]	Currentblockdeposits() may return an incorrect value	Low	Resolved

8. Findings

8.1. High Findings

[H-01] Unapproved requests are not deducted from total assets

Severity

Impact: Medium

Likelihood: High

Description

In the `HyperEvmVault` contract, the `requestRedeem()` function records the requested assets and shares but does not subtract these amounts from the total assets (`tvl()`) and total supply (`totalSupply()`). This leads to incorrect share price calculations until the withdrawal is finalized.

```
function requestRedeem(uint256 shares_) external nonReentrant {
        V1Storage storage $ = _getV1Storage();
        uint256 balance = this.balanceOf(msg.sender);
        // Determine if the user withdrawal request is valid
        require(shares_ <= balance, Errors.INSUFFICIENT_BALANCE());

        RedeemRequest storage request = $.redeemRequests[msg.sender];
        request.shares += shares_;
        require(request.shares <= balance, Errors.INSUFFICIENT_BALANCE());

        // User will redeem assets at the current share price
        uint256 tvl_ = _totalEscrowValue($);
        _takeFee($, tvl_);
        uint256 assetsToRedeem = shares_.mulDivDown(tvl_, totalSupply());

        request.assets += uint64(assetsToRedeem);
        $.totalRedeemRequests += uint64(assetsToRedeem);
        --snip--
    }
```

Example: Here's a comparison of the two scenarios (with and without adjusting totals):

1. **Initial State**:

   - Without Adjusting Totals: The total value locked (TVL) is 1,000,010 assets, and the total supply is 1,000,010 shares.
   - With Adjusting Totals: The TVL is 1,000,010 assets, and the total supply is 1,000,010 shares.

2. **User Requests Redemption**:

   - Without Adjusting Totals: A user requests to redeem 1,000,000 shares and 1,000,000 assets.
   - With Adjusting Totals: A user requests to redeem 1,000,000 shares and 1,000,000 assets.

3. **After Request**:

   - Without Adjusting Totals: The TVL and total supply remain unchanged at 1,000,010 assets and 1,000,010 shares, respectively.
   - With Adjusting Totals: The TVL is adjusted to 10 assets, and the total supply is adjusted to 10 shares.

4. **Vault Loses 5 Assets**:

   - Without Adjusting Totals: The TVL decreases to 1,000,005 assets, while the total supply remains at 1,000,010 shares.
   - With Adjusting Totals: The TVL decreases to 5 assets, and the total supply remains at 10 shares.

5. **Share Price Calculation**:

   - Without Adjusting Totals: The share price is calculated as approximately 1 (1,000,005 / 1,000,010).
   - With Adjusting Totals: The share price is calculated as 0.5 (5 / 10).

6. **Impact on Other Users**:

   - Without Adjusting Totals: Other users can deposit or withdraw at an incorrect share price (~1).
   - With Adjusting Totals: Other users can deposit or withdraw at the correct share price (0.5).

Recommendations

Subtract the requested redemption amounts from the total assets (`tvl()`) and total supply when a redemption request is made.

[H-02] Withdraw check can be bypassed

Severity

Impact: High

Likelihood: Medium

Description

In VaultEscrow, we will withdraw assets from the Vault in the core chain via function withdraw(). In withdraw() function, we have one security sanity check. This will make sure that we will not withdraw more asset than what we deposit in the vault.

But this security check can be bypassed. We get the vault's equity based on the `VAULT_EQUITY_PRECOMPILE_ADDRESS`.

Based on the HyperLiquid doc, the testnet EVM provides precompiles that allows querying HyperCore information. The precompile addresses start at 0x0000000000000000000000000000000000000800 and have methods for querying perps positions, spot balances, vault equity, staking delegations, oracle prices, and the L1 block number.

The values are guaranteed to match the latest HyperCore state at the time the EVM block is constructed.`

So when we start to create one new EVM block, we will sync the Core Chain's latest states to EVM chain's precompile contracts, including `VAULT_EQUITY_PRECOMPILE_ADDRESS`. In the same EVM block, Core Chain will not sync data per transaction. So it's quite possible that the data read from the `VAULT_EQUITY_PRECOMPILE_ADDRESS` is staled.

For example:

1. Alice deposit 1000 USDC via vault escrow 1.
2. Bob deposits 1000 USDC via vault escrow 2.

3. When the system constructs EVM block 100, we will sync the vault's equity to `VAULT_EQUITY_PRECOMPILE_ADDRESS`, escrow 1(1000), escrow 2(1000).
4. Bob requests withdraw from vault escrow 2 in block 1000.
5. Alice requests withdraw from vault escrow 2 in block 1000, this security check will be passed because the vault equity for escrow 2 will keep 1000. Alice's EVM transaction can pass. But this operation will fail in L1 chain.
6. Alice withdraws to get 1000 USDC.
7. Bob tries to withdraw, but Bob's transaction will be reverted because there are not enough assets in escrow 2.

```
address public constant VAULT_EQUITY_PRECOMPILE_ADDRESS = 0x0000000000000000000000000

    function withdraw(uint64 assets_) external override onlyVaultWrapper {
@>          require(assets_ <= _vaultEquity(), Errors.INSUFFICIENT_VAULT_EQUITY
  ());
        uint256 amountPerp = (_perpDecimals > _evmSpotDecimals)
            ? assets_ * (10 ** (_perpDecimals - _evmSpotDecimals))
            : assets_ / (10 ** (_evmSpotDecimals - _perpDecimals));

        L1_WRITE_PRECOMPILE.sendVaultTransfer(_vault, false, uint64
          (amountPerp));
        L1_WRITE_PRECOMPILE.sendUsdClassTransfer(uint64(amountPerp), false);
        L1_WRITE_PRECOMPILE.sendSpot
          (HYPERLIQUID_SPOT_BRIDGE, _assetIndex, assets_);
    }
```

```
function _vaultEquity() internal view returns (uint256) {
        (bool success, bytes memory result) =
            VAULT_EQUITY_PRECOMPILE_ADDRESS.staticcall(abi.encode(address
              (this), _vault));
        require(success, "VaultEquity precompile call failed");

        UserVaultEquity memory userVaultEquity = abi.decode(result,
          (UserVaultEquity));

        uint256 equityInSpot = (_perpDecimals > _evmSpotDecimals)
            ? userVaultEquity.equity / (10 **
              (_perpDecimals - _evmSpotDecimals))
            : userVaultEquity.equity * (10 **
              (_evmSpotDecimals - _perpDecimals));

        return equityInSpot;
    }
```

Recommendations

We need to add one internal accounting to record the latest vault equity. If this is the first transaction in this block, read from the precompile contract and store the vault equity. If this is the next transaction on this block, read the vault equity from the internal accounting.

[H-03] Incorrect `erc20` transfer implementation between evm and core chains

Severity

Impact: High

Likelihood: Medium

Description

In VaultEscrow, we will transfer users' deposit from EVM Chain to Core Chain and deposit into the related vault.

In function deposit(), we will transfer USDC tokens to precompiled system address `0x2222222222222222222222222222222222222222`. The system address will help us to transfer USDC tokens from EVM chain to the Core chain.

The problem here is that HyperLiquid changes the transfer method according to this.

Based on the latest doc, every token has a specific system address on the Core, which is the address with first byte 0x20 and the remaining bytes all zeros, except for the token index encoded in big-endian format. For example, for token index 200, the system address would be 0x20000000000000000000000000000000000000c8 .

```
/// @notice The address of the hyperliquid spot bridge, used for sending
    // tokens to L1.

         address public constant HYPERLIQUID_SPOT_BRIDGE = 0x2222222222222222222222222

    function deposit(uint64 amount) external onlyVaultWrapper {
        ERC20Upgradeable(_asset).safeTransfer(HYPERLIQUID_SPOT_BRIDGE, amount);
    }
```

So we should no use this legacy system address `0x2222222222222222222222222222222222222222`.

Recommendations

Refactor the implementation based on the latest doc. Transfer the related token to the related system address.

[H-04] Malicious users may block other withdrawals

Severity

Impact: High

Likelihood: Medium

Description

In HyperEvmVault, users need to take 2 steps to finish the redeem operation. Step1: request redeem. In this step, we will calculate the redeem amount according to current share price. Step2: redeem. We will deduct the actual redeem amount from the `$.redeemRequests[msg.sender]`.

The problem here is that when we deduct the assets, shares from `$.redeemRequests[msg.sender]`, we use one memory variable and fail to store the updated value into the storage. We fail to deduct the assets and shares from the `$.redeemRequests[msg.sender]`.

There are several impacts for this vulnerability:

1. Users fail to transfer their shares because the transfer check think users still have some redeem request.
2. Normal users' withdrawal can be blocked by malicious users.

For example:

1. Let's assume share's price is 1:1.
2. Alice deposits 2000 USDC.
3. Bob deposits 1000 USDC.
4. Bob requests redeem 1000 USDC.
5. Alice request redeem 1000 USDC.
6. Alice withdraws at the first time to get 1000 USDC.
7. Alice withdraws secondly to get another 1000 USDC.

8. When bob wants to withdraw, the withdraw will be reverted, because no funds exist in Escrow vault.

```
function _beforeWithdraw(uint256 assets_, uint256 shares_) internal {
        V1Storage storage $ = _getV1Storage();
@>        RedeemRequest memory request = $.redeemRequests[msg.sender];
        require(request.assets >= assets_, Errors.WITHDRAW_TOO_LARGE());
        require(request.shares >= shares_, Errors.WITHDRAW_TOO_LARGE());
        request.assets -= uint64(assets_);
        request.shares -= shares_;
        $.totalRedeemRequests -= uint64(assets_);
        _fetchAssets(assets_);
    }
```

```
function _beforeTransfer
      (address from_, address, /*to_*/ uint256 amount_) internal {
        V1Storage storage $ = _getV1Storage();
        uint256 balance = this.balanceOf(from_);
        RedeemRequest memory request = $.redeemRequests[from_];

        _takeFee($, _totalEscrowValue($));

        if (request.shares > 0) {
            require
              (balance - amount_ >= request.shares, Errors.TRANSFER_BLOCKED());
        }
    }
```

Recommendations

```
request.assets -= uint64(assets_);
         request.shares -= shares_;
+        $.redeemRequests[msg.sender] = request;
```

[H-05] User can drain escrow via approved withdrawals

Severity

Impact: High

Likelihood: Medium

Description

Whenever a user wishes to withdraw, they must create a request via the `HyperEvmVault::requestRedeem` function:

```
function requestRedeem(uint256 shares_) external nonReentrant {
    V1Storage storage $ = _getV1Storage();
    uint256 balance = this.balanceOf(msg.sender);
    // Determine if the user withdrawal request is valid
    require(shares_ <= balance, Errors.INSUFFICIENT_BALANCE());

    RedeemRequest storage request = $.redeemRequests[msg.sender];

    request.shares += shares_;
    require(request.shares <= balance, Errors.INSUFFICIENT_BALANCE());

    uint256 tvl_ = _totalEscrowValue($);
    _takeFee($, tvl_);
    uint256 assetsToRedeem = shares_.mulDivDown(tvl_, totalSupply());

    request.assets += uint64(assetsToRedeem);
    $.totalRedeemRequests += uint64(assetsToRedeem);
    emit RedeemRequested(msg.sender, shares_, assetsToRedeem);

    VaultEscrow escrowToRedeem = VaultEscrow($.escrows[redeemEscrowIndex()]);
    escrowToRedeem.withdraw(uint64(assetsToRedeem));
}
```

The function calculates the assets/share ratio and saves it into the user's request. When withdrawing, the `_beforeWithdraw` function is invoked:

```
function _beforeWithdraw(uint256 assets_, uint256 shares_) internal {
    V1Storage storage $ = _getV1Storage();

    RedeemRequest memory request = $.redeemRequests[msg.sender];
    require(request.assets >= assets_, Errors.WITHDRAW_TOO_LARGE());
    require(request.shares >= shares_, Errors.WITHDRAW_TOO_LARGE());

    request.assets -= uint64(assets_);
    request.shares -= shares_;
    $.totalRedeemRequests -= uint64(assets_);

    _fetchAssets(assets_);
}
```

This function deducts from the request of the `msg.sender` and fetches the assets from the escrow. However, `ERC4626::withdraw` also supports approvals, which a malicious actor could exploit to drain the vault equity of any escrow.

Proof of Concept (PoC)

Preconditions:

- **Vault equity of redeem escrow: 2000 USDC**
- **Vault equity of deposit escrow: 0**
- **Assets/Share ratio: 1:1**

Exploit Steps:

1. **Alice creates two addresses (Address1 and Address2) and mints 400 shares to each.**
2. **Alice grants Address1 an unlimited (`type(uint256).max`) approval over Address2's shares.**
3. **Alice initiates redeem requests from both addresses:**
    - **Redeem escrow vault equity: 1200 USDC**
    - **Deposit escrow vault equity: 800 USDC**
4. **Alice, using Address1, executes a withdrawal on behalf of Address2:**
    - `_beforeWithdraw` deducts assets from Address1's request but burns shares from Address2.
5. **Alice creates another redeem request, extracting another 400 USDC from the redeem escrow:**
    - **Redeem escrow equity: 800 USDC**
    - **Deposit Escrow equity: 800 USDC**
6. **400 USDC is now permanently locked, because of Address2's untouched request.**

By depositing only 800 USDC, Alice managed to extract 1200 USDC from the redeem escrow, successfully locking 400 USDC forever in the escrow contract.

Impact:

Alice can continuously repeat this exploit, siphoning assets from escrows and leaving a portion of funds permanently locked. This creates an imbalance where affected users are unable to withdraw their full entitlements.

Recommendations

Consider modifying `_beforeWithdraw` to explicitly pass `owner` as an argument and deduct from their request instead of `msg.sender` to prevent unauthorized withdrawals on behalf of another user.

8.2. Medium Findings

[M-01] `setManagementFeeBps()` fails to update fee state

Severity

Impact: Medium

Likelihood: Medium

Description

The `setManagementFeeBps()` function in the `HyperEvmVault` contract allows the owner to update the management fee rate. However, it does not update the fee state (e.g., `lastFeeCollectionTimestamp`) when the fee rate is changed. This can lead to incorrect fee calculations, especially in less active vaults.

```
function setManagementFeeBps
      (uint64 newManagementFeeBps_) external onlyOwner {
        require(newManagementFeeBps_ <= BPS_DENOMINATOR, Errors.FEE_TOO_HIGH());
        _getV1Storage().managementFeeBps = newManagementFeeBps_;
    }
```

If the fee rate was previously set to 0 for an extended period (e.g., 1 year to attract users), the `lastFeeCollectionTimestamp` could be set in a long time ago. After increasing the fee rate (e.g., to 0.01%), the contract may incorrectly calculate fees based on the entire time elapsed since the outdated `lastFeeCollectionTimestamp`, leading to excessive fee collection.

Recommendations

Update the `setManagementFeeBps()` function to take the fee and reset the `lastFeeCollectionTimestamp` when the fee rate is changed.

[M-02] Incorrect rounding in `mint()`

Severity

Impact: Medium

Likelihood: Medium

Description

In the `HyperEvmVault` contract, the `mint()` function calculates the number of assets by rounding down:

```
function mint(uint256 shares, address receiver) public override
      (ERC4626Upgradeable, IERC4626) nonReentrant returns (uint256 assets) {
        V1Storage storage $ = _getV1Storage();

        if (totalSupply() == 0) {
            // If the vault is empty then we need to initialize last fee
            // collection timestamp
            assets = shares;
            $.lastFeeCollectionTimestamp = uint64(block.timestamp);
        } else {
            uint256 tvl_ = _totalEscrowValue($);
            _takeFee($, tvl_);
            assets = shares.mulDivDown(tvl_, totalSupply());
        }
       --snip--
    }
```

This implementation has two critical issues:

1. Violation of ERC4626 Standard: The ERC4626 standard expects rounding up for `mint` function.

2. Exploitation Risk: A malicious user can exploit this rounding-down mechanism to deposit the minimum amount of assets while minting more shares. This could allow the attacker to steal funds from other users.

Recommendations

Round up to calculate amount of assets in `mint()` function.

[M-03] Wrong fee share calculations in `_takeFee()`

Severity

Impact: Medium

Likelihood: Medium

Description

In the `HyperEvmVault` contract, the `_takeFee()` function is used to calculate and collect management fees from the vault. The function calculates the fee amount and mints corresponding shares to the `feeRecipient` to deduct the fee from **depositors**.

```
function _takeFee(V1Storage storage $, uint256 grossAssets) private returns
  (uint256) {
    uint256 feeTake_ = _calculateFee($, grossAssets);

    // Only update state if there's a fee to take
    if (feeTake_ > 0) {
        $.lastFeeCollectionTimestamp = uint64(block.timestamp);
        uint256 sharesToMint = _convertToShares(feeTake_, Math.Rounding.Floor);
        _mint($.feeRecipient, sharesToMint);
    }
    return feeTake_;
}

function _convertToShares(
  uint256assets,
  Math.Rounding/*rounding*/
) internal view override returns (uint256
    return assets.mulDivDown(totalSupply(), tvl());
}
```

The issue arises in using `_convertToShares()` function to calculate the fee shares. It uses the total assets (`tvl()`) which includes the current fee assets, meaning the fee is taken from already collected fee assets too. As a result, the fee taken will be less than the intended amount.

Recommendations

To calculate new fee shares to mint, consider the total fee amount and total fee shares. Code should use `tvl() - feeTake_` as total assets when calculating fee share.

[M-04] Protocol fee rounding can be weaponized for minimal earnings

Severity

Impact: High

Likelihood: Low

Description

In `_previewFeeShares` and `_takeFee`, we can see that after calculating the fee assets that the protocol is entitled to, a conversion of the assets is made to fees, via the `_convertToShares` functions. Observe the specified, albeit unused rounding direction.

```
function _previewFeeShares
      (V1Storage storage $, uint256 tvl_) internal view returns (uint256) {
        uint256 expectedFee = _calculateFee($, tvl_);
@>1     return _convertToShares(expectedFee, Math.Rounding.Floor);
    }

    /**
     * @notice Takes the management fee from the vault
     * @dev There is a 0.015% annual management fee on the vault's total assets.
     * @param grossAssets The total value of the vault
     * @return The amount of fees to take in underlying assets
     */
    function _takeFee
      (V1Storage storage $, uint256 grossAssets) private returns (uint256) {
        uint256 feeTake_ = _calculateFee($, grossAssets);

        // Only update state if there's a fee to take
        if (feeTake_ > 0) {
            $.lastFeeCollectionTimestamp = uint64(block.timestamp);
@>2         uint256 sharesToMint = _convertToShares
  (feeTake_, Math.Rounding.Floor);
            _mint($.feeRecipient, sharesToMint);
        }
        return feeTake_;
    }
```

`_convertToShares` performs the `mulDivDown` operation on returned fee assets to return the shares the protocol is entitled to.

```
function _convertToShares(
      uint256assets,
      Math.Rounding/*rounding*/
    ) internal view override returns (uint256
@>3     return assets.mulDivDown(totalSupply(), tvl());
    }
```

And this fee asset is calculated, time-based in `_calculateFee` as below:

```
function _calculateFee(
      V1Storagestorage$,
      uint256grossAssets
    ) internal view returns (uint256 feeAmount_
        if
          (grossAssets == 0 || block.timestamp <= $.lastFeeCollectionTimestamp) {
            return 0;
        }

        // Calculate time elapsed since last fee collection
@>4     uint256 timeElapsed = block.timestamp - $.lastFeeCollectionTimestamp;

        // We subtract the pending redemption requests from the total asset
        // value to avoid taking more fees than needed from
        //    users who do not have any pending redemption requests
        uint256 eligibleForFeeTake = grossAssets - $.totalRedeemRequests;
        // Calculate the pro-rated management fee based on time elapsed
@>5
       feeAmount_ = eligibleForFeeTake * $.managementFeeBps * timeElapsed / BPS_DENOMI

        return feeAmount_;
    }
```

So, in a dedicated attack (e.g via a script), in which a function that triggers `_takeFee` (e.g `transfer`) is repeatedly called every certain `timeElapsed` from `lastFeeCollectionTimestamp` (e.g 10 seconds) such that the returned value of `feeAmount_` is greater than 0 but still small enough that converting it to shares i.e multiplying it by `totalSupply()` will be less than `tvl()` and due to `mulDivDown` will round down to 0. Even though no fee shares are minted, the `lastFeeCollectionTimestamp` will still be updated, making a repetition of the attack viable. Note also legitimate transactions that occur between this `timeElapsed` also trigger this situation.

```
function _takeFee
      (V1Storage storage $, uint256 grossAssets) private returns (uint256) {
        uint256 feeTake_ = _calculateFee($, grossAssets);

        // Only update state if there's a fee to take
        if (feeTake_ > 0) {
@>6         $.lastFeeCollectionTimestamp = uint64(block.timestamp);
            uint256 sharesToMint = _convertToShares
              (feeTake_, Math.Rounding.Floor);
            _mint($.feeRecipient, sharesToMint);
        }
        return feeTake_;
    }
```

Recommendations

1. As a rule in DeFi, protocol fees should always round up in its favour.
2. If the recommendation above is not to be implemented, `lastFeeCollectionTimestamp` can be updated instead but only if `sharesToMint` > 0.

8.3. Low Findings

[L-01] Add checks for zero returned shares

Due to the round down in the deposit function, if a user gets frontrun my a large depositor or a whale, or if deposited assets is > `minDepositAmount` but its product with `totalSupply` is less than tvl, a user may end up receiving 0 shares for deposited assets, leading to fund loss.

```
} else {
            uint256 tvl_ = _totalEscrowValue($);
            _takeFee($, tvl_);
@>1         shares = assets.mulDivDown(totalSupply(), tvl_);
        }
```

[L-02] `maxwithdraw()` ignores pending redemptions

The `HyperEvmVault` contract inherits from `ERC4626Upgradeable` and implements the `maxWithdraw()` function, which calculates the maximum amount of assets a user can withdraw. However, the function does not account for pending redemption requests, leading to an overestimation of the available assets for withdrawal:

```
function maxWithdraw(address owner) public view virtual returns (uint256) {
        return _convertToAssets(balanceOf(owner), Math.Rounding.Floor);
    }

    function _convertToAssets(
      uint256shares,
      Math.Rounding/*rounding*/
    ) internal view override returns (uint256
        return shares.mulDivDown(tvl(), totalSupply());
    }
```

To address this issue, subtract the user's shares in pending redemption requests from their total share balance when calculating `maxWithdraw()`. This ensures that only the truly available shares are considered for withdrawal.

[L-03] Preview functions fail with supply=0

In `HyperEvmVault` contract, `previewDeposit` and `previewMint` functions does not handle supply=0 (return wrong number or revert)

```
function previewDeposit(uint256 assets_) public view override
      (ERC4626Upgradeable, IERC4626) returns (uint256) {
        V1Storage storage $ = _getV1Storage();
        uint256 tvl_ = _totalEscrowValue($);
        uint256 feeShares = _previewFeeShares($, tvl_);
        uint256 adjustedSupply = totalSupply() + feeShares;
        return assets_.mulDivDown(adjustedSupply, tvl_);
    }

    function previewMint(uint256 shares_) public view override
      (ERC4626Upgradeable, IERC4626) returns (uint256) {
        V1Storage storage $ = _getV1Storage();
        uint256 tvl_ = _totalEscrowValue($);
        uint256 feeShares = _previewFeeShares($, tvl_);
        uint256 adjustedSupply = totalSupply() + feeShares;
        return shares_.mulDivDown(tvl_, adjustedSupply);
    }
```

According to ERC4626 these functions should simulate the effects of their deposit/mint at the current block. As we see in `deposit` function, it should return amount of assets when `supply = 0`

```
function deposit(uint256 assets, address receiver) public override
  (ERC4626Upgradeable, IERC4626) nonReentrant returns (uint256 shares) {
        V1Storage storage $ = _getV1Storage();
        require(assets >= $.minDepositAmount, Errors.MIN_DEPOSIT_AMOUNT());

        if (totalSupply() == 0) {
            // If the vault is empty then we need to initialize last fee
            // collection timestamp
@>          shares = assets;
            $.lastFeeCollectionTimestamp = uint64(block.timestamp);
        } else {
            uint256 tvl_ = _totalEscrowValue($);
            _takeFee($, tvl_);
            shares = assets.mulDivDown(totalSupply(), tvl_);
        }
        --snip--
}
```

Add a condition to handle `supply=0` in preview functions.

[L-04] Protocol may charge more fees than expected

In HyperEvmVault, the protocol will charge some protocol fees based on the total assets we manage. If the calculated fee amount is zero, we will skip the fee process. The problem here is that we don't update the `lastFeeCollectionTimestamp` in this case. This will cause that we will use the incorrect time slot when we calculate protocol fees next time.

For example:

1. Alice deposits 1000 USDC in timestamp X.
2. Alice requests withdraw 1000 USDC in timestamp X + 100. There are some collect fees between X and X + 100.
3. Alice deposits 1000 USDC in timestamp X + 200. When we calculate the fee between X + 100 and X + 200, the fees are zero. Note we will not update the timestamp. Current `lastFeeCollectionTimestamp` is timestamp X + 100.
4. Bob deposits 1000 USDC in timestamp X + 300. When we calculate fees, we will calculate fees between X + 100 and X + 300 based on the asset amount 1000. We will charge more fees than expected.

```
function _takeFee
      (V1Storage storage $, uint256 grossAssets) private returns (uint256) {
        uint256 feeTake_ = _calculateFee($, grossAssets);

        if (feeTake_ > 0) {
            $.lastFeeCollectionTimestamp = uint64(block.timestamp);
            uint256 sharesToMint = _convertToShares
              (feeTake_, Math.Rounding.Floor);
            _mint($.feeRecipient, sharesToMint);
        }
        return feeTake_;
    }
```

When we calculate and take the fees, update the `lastFeeCollectionTimestamp` timely.

[L-05] `Currentblockdeposits()` may return an incorrect value

In the case when the L1 block changes, `HyperEvmVault::currentBlockDeposits` function may return the deposits of the previous block if it is called before the storage `currentBlockDeposits` value is updated.

```
function currentBlockDeposits() external view override returns (uint64) {
        return _getV1Storage().currentBlockDeposits;
    }
```

Consider return 0 if the current L1 block is different from lastL1Block.

```
function currentBlockDeposits() external view override returns (uint64) {
+       return _getV1Storage().lastL1Block == l1Block() ? _getV1Storage
+ ().currentBlockDeposits : 0;
-       return _getV1Storage().currentBlockDeposits;
    }
```