Blueberry Security Review

Pashov Audit Group

Conducted by: zark, pontifex, adriro, Ragnarok, ph, Atharv

March 26th 2025 - March 29th 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 - 216e59ab4f4f8e46bd337cc8fc934dce86ceed40

fixes review commit hash - 1844c2bfbc787628eb8d858b9d764ca3483d571d

Scope

The following smart contracts were in scope of the audit:

- `HyperEvmVault`
- `VaultEscrow`

7. Executive Summary

Over the course of the security review, zark, pontifex, adriro, Ragnarok, ph, Atharv engaged with Blueberry to review Blueberry. In this period of time a total of **8** issues were uncovered.

Protocol Summary

Protocol Name	Blueberry
Repository	https://github.com/Blueberryfi/blueberry-v2-contracts
Date	March 26th 2025 - March 29th 2025
Protocol Type	Lending

Findings Count

Severity	**Amount**
Critical	1
High	3
Low	4
Total Findings	**8**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Incorrect fee due to double subtracting requestSum.assets	Critical	Resolved
[H-01]	Flawed withdrawal logic when caller differs from share owner	High	Resolved
[H-02]	Donated tokens are never deposited into vaults	High	Resolved
[H-03]	Precision loss in _withdrawFromL1Vault() can cause asset transfer reverts	High	Resolved
[L-01]	Incorrect max withdrawable assets amount causes unexpected revert	Low	Resolved
[L-02]	First depositor attack is possible in HyperEvmVault	Low	Acknowledged
[L-03]	Griefing attack can block users from redeeming requests	Low	Acknowledged
[L-04]	Incorrect address check in constructor()	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Incorrect fee due to double subtracting `requestSum.assets`

Severity

Impact: High

Likelihood: High

Description

The `_calculateFee` function in the `HyperEvmVault` contract incorrectly subtracts `$.requestSum.assets` from `grossAssets`, even though this subtraction has already been performed upstream in `_totalEscrowValue`. This results in an underestimation of `eligibleForFeeTake`, leading to incorrect fee calculations.

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

        // Calculate time elapsed since last fee collection
        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 - $.requestSum.assets;
        // Calculate the pro-rated management fee based on time elapsed

                  feeAmount_ = eligibleForFeeTake * $.managementFeeBps * timeElapsed /

        return feeAmount_;
    }
```

However, `grossAssets` is passed directly from `_totalEscrowValue`, which already does the substraction:

```
function _totalEscrowValue(V1Storage storage $) internal view returns
      (uint256 assets_) {
        uint256 escrowLength = $.escrows.length;
        for (uint256 i = 0; i < escrowLength; ++i) {
            VaultEscrow escrow = VaultEscrow($.escrows[i]);
            assets_ += escrow.tvl();
        }

        if ($.lastL1Block == l1Block()) {
            assets_ += $.currentBlockDeposits;
        }

@>          return assets_ - $.requestSum.assets;
    }
```

So, subtracting `$.requestSum.assets` again, results in fees being calculated on a doubly reduced value, or even underflow and revert can happen.

Recommendations

Consider updating `_calculateFee` and not subtract `$.requestSum.assets` again.

8.2. High Findings

[H-01] Flawed withdrawal logic when caller differs from share owner

Severity

Impact: Medium

Likelihood: High

Description

Withdrawals in HyperEvmVault have a custom behavior. Users need to first call `requestRedeem()` to move funds from L1 back to the EVM side. After this, available funds to be withdrawn are stored in the `redeemRequests` mapping.

These mechanics lead to a specialization of the `previewWithdraw()` and `previewRedeem()` functions. Instead of operating with the global supply or TVL values, these functions are implementended using the `redeemRequests` mapping:

```
function previewWithdraw(uint256 assets_) public view override
  (ERC4626Upgradeable, IERC4626) returns (uint256) {
    V1Storage storage $ = _getV1Storage();
    RedeemRequest memory request = $.redeemRequests[msg.sender];
    return assets_.mulDivUp(request.shares, request.assets);
}

function previewRedeem(uint256 shares_) public view override
  (ERC4626Upgradeable, IERC4626) returns (uint256) {
    V1Storage storage $ = _getV1Storage();
    RedeemRequest memory request = $.redeemRequests[msg.sender];
    return shares_.mulDivDown(request.assets, request.shares);
}
```

Note that this forces the implementation to predicate over an account, which is chosen here to be `msg.sender`.

Withdrawals in ERC4626 also support a flow in which the caller could be different from the owner of the shares, leveraging ERC20 approvals. While

this is correctly implemented in the overridden version of `_withdraw()`, the implementation still uses the preview functions that operate on `msg.sender`. This means that conversion will be calculated using the caller balances, but balances will be modified for the owner account, leading to accounting issues and a potential denial of service.

Recommendations

Override the implementations of `withdraw()` and `redeem()` so that the `previewWithdraw()` and `previewRedeem()` actions execute the conversion on the owner (i.e. by using `$.redeemRequests[owner]`).

[H-02] Donated tokens are never deposited into vaults

Severity

Impact: Medium

Likelihood: High

Description

The `HyperEvmVault` contract has to use the asset balances of the `VaultEscrow`s in the share's price calculation to handle tokens being received from the system contract. In normal the balances should not exceed the `$.requestSum.assets` value. The problem is that in case of donation to the vaults, the excess balance increases the shares price but can not be distributed, i.e. it is locked on the vaults. This way, users can not redeem all shares. Also, the excess amount increases the management fee. Assume an attacker deposited 100 tokens and donated another 100 tokens. Then the attacker has 100 shares and the total amount of assets locked by the vault equals 200 since the `$.requestSum.assets` is zero. HyperEvmVault.sol

```
function _totalEscrowValue(V1Storage storage $) internal view returns
      (uint256 assets_) {
        uint256 escrowLength = $.escrows.length;
        for (uint256 i = 0; i < escrowLength; ++i) {
            VaultEscrow escrow = VaultEscrow($.escrows[i]);
>>          assets_ += escrow.tvl();
        }

        if ($.lastL1Block == l1Block()) {
            assets_ += $.currentBlockDeposits;
        }

        return assets_ - $.requestSum.assets;
    }
```

VaultEscrow.sol

```
function tvl() public view returns (uint256) {
        uint256 assetBalance = ERC20Upgradeable(_asset).balanceOf(address
          (this));
        (uint64 vaultEquity_,) = _vaultEquity();
        return uint256(vaultEquity_) + assetBalance;
    }
```

Then an innocent user mints 100 shares. Now the total supply is 200 and the total amount of assets is 400. Then the attacker requests redeeming of 100 shares. Now the total amount of assets is 200 because the `$.requestSum.assets` is 200.
Then the user tries to request redeeming 100 shares but it will always revert because the vaults equity does not include the donated tokens:

```
function withdraw(uint64 assets_) external override onlyVaultWrapper {
>>      (uint64 vaultEquity_, uint64 lockedUntilTimestamp_) = _vaultEquity();
        require(block.timestamp > lockedUntilTimestamp_, Errors.L1_VAULT_LOCKED
          ());

        // Update the withdraw state for the current L1 block
        L1WithdrawState storage l1WithdrawState_ = _getV1Storage
          ().l1WithdrawState;
        _updateL1WithdrawState(l1WithdrawState_);
>>      l1WithdrawState_.lastWithdraws += assets_;

        // Ensure we havent exceeded requests for the current L1 block
>>      require(
  vaultEquity_>=l1WithdrawState_.lastWithdraws,
  Errors.INSUFFICIENT_VAULT_EQUITY
)

        // Withdraw from L1 vault
        _withdrawFromL1Vault(assets_);
    }
<..>
    function _vaultEquity() internal view returns (uint64, uint64) {
        (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 = _scaleToSpotDecimals(userVaultEquity.equity);

        return (uint64(equityInSpot), userVaultEquity.lockedUntilTimestamp);
    }
```

Recommendations

Consider depositing any excess amounts in the L1 vault.

[H-03] Precision loss in `_withdrawFromL1Vault()` can cause asset transfer reverts

Severity

Impact: High

Likelihood: Medium

Description

In the `VaultEscrow::_withdrawFromL1Vault()` function, the withdrawal amount from the L1 vault can be **less than** the amount sent back to the EVM spot (when both are transformed to floating-point numbers). This discrepancy can cause the transaction on Hyperliquid L1, which sends assets back to the EVM spot, to revert, while the transaction on Hyper EVM to request redeem (`HyperEvmVault::requestRedeem()`) has already succeeded.

As a result, the redeem request has already succeeded, but the user cannot withdraw assets because they were not transferred to `VaultEscrow`.

`VaultEscrow::_withdrawFromL1Vault()` function:

```
function _withdrawFromL1Vault(uint64 assets_) internal {
    uint256 amountPerp = _scaleToPerpDecimals(assets_);
    // Withdraws assets from L1 vault
    L1_WRITE_PRECOMPILE.sendVaultTransfer(_vault, false, uint64(amountPerp));
    // Transfer assets to L1 spot
    L1_WRITE_PRECOMPILE.sendUsdClassTransfer(uint64(amountPerp), false);
    // Bridges assets back to escrow's EVM account
=>  L1_WRITE_PRECOMPILE.sendSpot(assetSystemAddr(), _assetIndex, assets_);
}
```

Specifically, the withdrawal amount from the L1 vault is `amountPerp`, the amount sent back to the EVM spot is `assets_` and the conversion is done as `amountPerp = _scaleToPerpDecimals(assets_)`. If `_perpDecimals` is less than `_evmSpotDecimals`, precision loss occurs, which may lead to a revert when sending assets back to the EVM spot. Example case:

Assume:

- `_perpDecimals = 6`
- `_evmSpotDecimals = 8`
- `assets_ = 100001111`

Then:

- `amountPerp = assets_ / 10^2 = 1000011`
- When `amountPerp` is transformed back to spot decimals, the transformed value is `100001100`, which is less than `100001111`.
- As a result, the transaction on Hyperliquid L1 to send assets back to the EVM spot reverts.

Recommendations

Modify the `VaultEscrow::_withdrawFromL1Vault()` function to verify that `amountPerp` when transformed back to spot decimals is equal to `assets_`, or only send back the transformed spot-decimal amount.

```
function _withdrawFromL1Vault(uint64 assets_) internal {
   uint256 amountPerp = _scaleToPerpDecimals(assets_);
   // Withdraws assets from L1 vault
   L1_WRITE_PRECOMPILE.sendVaultTransfer(_vault, false, uint64(amountPerp));
   // Transfer assets to L1 spot
   L1_WRITE_PRECOMPILE.sendUsdClassTransfer(uint64(amountPerp), false);
   // Bridges assets back to escrow's EVM account
-   L1_WRITE_PRECOMPILE.sendSpot(assetSystemAddr(), _assetIndex, assets_);
+   L1_WRITE_PRECOMPILE.sendSpot(assetSystemAddr
+ (), _assetIndex, _scaleToSpotDecimals(amountPerp));
}
```

8.3. Low Findings

[L-01] Incorrect max withdrawable assets amount causes unexpected revert

The `HyperEvmVault.maxWithdrawableAssets` function returns the max amount of assets that can be requested to be withdrawn, i.e. which is controlled by the vault. When users invoke the `requestRedeem` function, the sum of requested amounts in the current block is increased. The sum should not exceed the vault equity, which will be updated only on the next block.

```
function withdraw(uint64 assets_) external override onlyVaultWrapper {
        (uint64 vaultEquity_, uint64 lockedUntilTimestamp_) = _vaultEquity();
        require(block.timestamp > lockedUntilTimestamp_, Errors.L1_VAULT_LOCKED
          ());

        // Update the withdraw state for the current L1 block
        L1WithdrawState storage l1WithdrawState_ = _getV1Storage
          ().l1WithdrawState;
>>      _updateL1WithdrawState(l1WithdrawState_);
        l1WithdrawState_.lastWithdraws += assets_;

        // Ensure we havent exceeded requests for the current L1 block
        require(
          vaultEquity_>=l1WithdrawState_.lastWithdraws,
          Errors.INSUFFICIENT_VAULT_EQUITY
        )

        // Withdraw from L1 vault
        _withdrawFromL1Vault(assets_);
    }
<..>
    function _updateL1WithdrawState
      (L1WithdrawState storage l1WithdrawState_) internal {
        uint64 currentL1Block = l1Block();
        if (currentL1Block > l1WithdrawState_.lastWithdrawBlock) {
            l1WithdrawState_.lastWithdrawBlock = currentL1Block;
            l1WithdrawState_.lastWithdraws = 0;
        }
    }
```

Users can use the `maxWithdrawableAssets` function to control parameters for the `requestRedeem` invoke. But the return value is not decreased by `l1WithdrawState_.lastWithdraws` value when `currentL1Block == l1WithdrawState_.lastWithdrawBlock`.

```
function maxWithdrawableAssets() public view returns (uint256) {
        V1Storage storage $ = _getV1Storage();
        VaultEscrow escrowToRedeem = VaultEscrow($.escrows[redeemEscrowIndex
          ()]);
        return escrowToRedeem.vaultEquity();
    }
```

This way users can request for redemption more assets than can be redeemed from the vault and face unexpected error.

Consider taking into account the `l1WithdrawState_.lastWithdraws` value when `currentL1Block == l1WithdrawState_.lastWithdrawBlock` for the available amount of assets calculation.

[L-02] First depositor attack is possible in `HyperEvmVault`

The `HyperEvmVault` is vulnerable to a classic first depositor attack due to the way share-to-asset ratios are initialized when the vault has low or no supply.

Steps:

1. Victim is preparing to deposit 10e8 USDC.
2. Attacker front-runs and deposits 10e8 USDC. Since this is the first deposit, he receives 10e8 shares. Now, `shares == totalAssets == 10e18`.
3. Attacker withdraws all shares except 1 wei — leaving 1 share in the vault, which now holds 1 wei USDC. As we can see, the `totalAssets` and `totalSupply` will reflect this redemption, even if it is asynchronous. Now `shares == totalAssets == 1`

```
function totalSupply() public view override
      (ERC20Upgradeable, IERC20) returns (uint256) {
        V1Storage storage $ = _getV1Storage();
@>        return super.totalSupply() - $.requestSum.shares;
    }

    function _totalEscrowValue(V1Storage storage $) internal view returns
      (uint256 assets_) {
        uint256 escrowLength = $.escrows.length;
        for (uint256 i = 0; i < escrowLength; ++i) {
            VaultEscrow escrow = VaultEscrow($.escrows[i]);
            assets_ += escrow.tvl();
        }

        if ($.lastL1Block == l1Block()) {
            assets_ += $.currentBlockDeposits;
        }

@>        return assets_ - $.requestSum.assets;
    }
```

4. Attacker directly transfers 5e8 USDC in the escrow. Now `shares = 1, totalAssets = 5e8 + 1`.
5. First depositor victim deposits 10e8 USDC now, and he gets minted `10e8 * 1 /(5e8+1) = 1.99 = 1 share`. Now, `shares = 2, totalAssets = 15e8 + 1`.
6. Attacker withdraws his 1 share and takes 7.5e8 USDC, profiting `7.5e8 - 5e8 = 2.5e8` USDC profit.]

In these 6 steps, victim has lost `2.5e8 USDC` which the attacker profited and an inflation attack successfully took place.

Consider making the first deposit yourself or follow any other of these battle tested solutions.

[L-03] Griefing attack can block users from redeeming requests

The daily max amount of assets that can be requested to be withdrawn is capped by the equity of solo escrow contract:

```
function requestRedeem(uint256 shares_) external nonReentrant {
<..>
>>      VaultEscrow escrowToRedeem = VaultEscrow($.escrows[redeemEscrowIndex
  ()]);
        escrowToRedeem.withdraw(uint64(assetsToRedeem));
    }
<..>
    function redeemEscrowIndex() public view returns (uint256) {
        uint256 len = _getV1Storage().escrows.length;
        if (len == 1) {
            return 0;
        }

        uint256 depositIndex = depositEscrowIndex();
>>      return (depositIndex + 1) % len;
    }
```

This means that if someone has requested the whole equity of the `escrowToRedeem` contract, the next request can be executed only on the next day.

Suppose there are 7 escrows in the protocol with a near value of equity in each. Then a malicious user can prevent other users from requesting redemption using a token amount which is seven times less than the total amount of assets locked by the vault. For this purpose the attacker can once a day request whole daily equity and redeposit the amount again in the next block. Another way of the attack is exploiting the lack of a cooldown period in the protocol. Users can request redemption immediately after depositing with the only condition that the `escrowToRedeem` contract has enough equity.

This way the attacker can control the request functionality just for gas.

Consider implementing a cooldown period for requests and transfers.

[L-04] Incorrect address check in `constructor()`

In the `VaultEscrow::constructor()` function, the logical OR (`||`) operator is incorrectly used when checking that multiple addresses are not `address(0)`. With the current logic, the constructor only requires **at least one** of the three addresses to be non-zero, which is incorrect.

`VaultEscrow::constructor()` function:

```
constructor(
  addresswrapper_,
  addressvault_,
  addressasset_,
  uint64assetIndex_,
  uint8assetPerpDecimals_
) {
=>  require(wrapper_ != address(0) || vault_ != address(0) || asset_ != address
  (0), Errors.ADDRESS_ZERO());
    ...
}
```

Modify the condition to use the logical AND (`&&`) operator instead:

```
constructor(
  addresswrapper_,
  addressvault_,
  addressasset_,
  uint64assetIndex_,
  uint8assetPerpDecimals_
) {
-   require(wrapper_ != address(0) || vault_ != address(0) || asset_ != address
- (0), Errors.ADDRESS_ZERO());
+   require(wrapper_ != address(0) && vault_ != address(0) && asset_ != address
+ (0), Errors.ADDRESS_ZERO());
    ...
}
```