SHERLOCK

Security Review For Prodigy Finance

Collaborative Audit Prepared For:	**Prodigy Finance**
Lead Security Expert(s):	**deth**
	Obsidian
Date Audited:	**September 29 - October 8, 2025**

Introduction

Prodigy.Fi is a fully onchain platform for Dual Currency Investment (DCI) vaults, offering fixed yields in any market through a permissionless, non-custodial vault marketplace. With Prodigy.Fi, users can lock in guaranteed returns at subscription, accumulate BTC or ETH at target prices, and put idle stablecoins to work with passive yield strategies.

Scope

Repository: prodigyfi/brt-dci-contracts

Audited Commit: b65f93ce1d1635768d73d26e4ab216b3d67edc12

Final Commit: 18b27888651c6fe12d58c12bb1085f7330b5e42d

Files:

- src/CollateralPoolV2.sol
- src/FactoryCore.sol
- src/Factory.sol
- src/libraries/CalcLib.sol
- src/libraries/StructsLib.sol
- src/Router.sol
- src/VaultBatchManager.sol
- src/VaultCore.sol
- src/Vault.sol
- src/VaultV2.sol

Final Commit Hash

18b27888651c6fe12d58c12bb1085f7330b5e42d

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.

- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	2	3

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: Since CollateralPoolV2 does not support partial withdrawals, minor LP shortfalls can fully block user withdrawals [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-09-prodigy-finance-sept-25th/issues/12

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The CollateralPoolV2 contract does not support partial withdrawals, causing full withdrawal attempts to revert even when only a small portion of the amount is unavailable.

Vulnerability Detail

If an LP in CollateralPoolV2 has almost, but not entirely enough funds, to cover a user's withdrawal, the transaction will revert.

This occurs because `_updateLpBalance` enforces a strict require check that the LP's `ownDeposit` must meet the withdrawal amount. As a result, users with large positions may be unable to withdraw any funds when there is only a small shortfall.

This issue can only occur with whitelisted LP's, making it a very unlikely scenario.

Impact

A minor shortfalll from the LP will completely block user withdrawals, leaving user funds inaccessible until the LP replenishes the missing balance, if the LP does not fill the shortfall, users will permanently lose funds.

Code Snippet

https://github.com/sherlock-audit/2025-09-prodigy-finance-sept-25th/blob/3ce9fb38cffe4a26e601e2d556b56b201a2a513b/brt-dci-contracts/src/CollateralPoolV2.sol#L277-L283

Proof of Concept

Add the following file to `brt-dci-contracts/test/POC.t.sol`

The test `test_Shortfall_POC` shows that a withdrawal of over 250 USDC is blocked by a tiny shortfall of 0.1 USDC

```
pragma solidity ^0.8.26;

// SPDX-License-Identifier: UNLICENSED

import "forge-std/Test.sol";
import "forge-std/console.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {UnsafeUpgrades} from "openzeppelin-foundry-upgrades/Upgrades.sol";
import {SafeCast} from "@openzeppelin/contracts/utils/math/SafeCast.sol";
import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import {MessageHashUtils} from
↪ "@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";
import {VaultCoreErrors, VaultV2Errors, CollateralPoolErrors, CommonErrors} from
↪ "../src/libraries/ErrorsLib.sol";
import {CalcLib} from "../src/libraries/CalcLib.sol";
import {Factory} from "../src/Factory.sol";
import {Router} from "../src/Router.sol";
import {VaultV2} from "../src/VaultV2.sol";
import {CollateralPoolV2} from "../src/CollateralPoolV2.sol";
import {ICollateralPoolV2} from "../src/interfaces/ICollateralPoolV2.sol";
import {IVaultCore} from "../src/interfaces/IVaultCore.sol";
import {LpBalance} from "../src/libraries/StructsLib.sol";
import {
    CreateVaultParams,
    VaultParams,
    FeeParams,
    GetPriceOptions,
    LpBalance,
    VaultInfo
} from "../src/libraries/StructsLib.sol";
import {TestData} from "./TestData.sol";
import {MockERC20} from "./mock/MockERC20.sol";
import {MockChainlinkOracle} from "./mock/MockChainlinkOracle.sol";
import {MockPythOracle} from "./mock/MockPythOracle.sol";
import {PythAggregator} from "../src/PythAggregator.sol";

contract POC is Test, TestData {
    using SafeERC20 for IERC20;

    TestDataStruct private vaultTestData = aeroUsdcTestData;

    CollateralPoolV2 public collateralPoolV2;
    VaultV2 public vaultV2;

    IERC20 private baseToken = IERC20(AEROToken);
    IERC20 private quoteToken = IERC20(USDCToken);

    GetPriceOptions private getPriceOptions = GetPriceOptions({
        pythPublishTime: uint64(vaultTestData.expiry),
        pythMinConfidenceRatio: 1e18,
        chainlinkUseLatestAnswer: false,
        chainlinkRoundId: 0
    });

    uint256 oraclePriceAtCreation = 0.4e6; // 1 AERO = 0.4 USDC, pre-converted to
    ↪  USDC decimals

    address private collateralPoolV2Proxy;

    // Deposit with signature
    uint256 immutable signerPrivateKey =
    ↪  0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80;
    address immutable signer = vm.addr(signerPrivateKey);

    uint256 private nonce = 1;
    uint256 private deadline;

    receive() external payable {}

    function setUp() public {
        string memory rpcUrl = "https://1rpc.io/base";
        vm.createSelectFork(rpcUrl);
        vm.warp(vaultTestData.timestamp);

        // Mock the oracle that Pyth network provides
        mockPythOracle = new MockPythOracle(0.4e8); // 1 AERO = 0.5 USDC

        // Mock Chainlink Oracle
        mockChainlinkOracle = new
        ↪  MockChainlinkOracle(int256(vaultTestData.linkedOraclePrice));

        setUpRouter();
        setUpPythAggregator();
        setUpFactory(false, false);
        setUpFunding();

        // Deploy VaultV2 implementation
        address vaultV2Implementation = address(new VaultV2());
        factory.setVaultImplementation(vaultV2Implementation, 2);

        address collateralPoolV2Impl = address(new CollateralPoolV2());
        collateralPoolV2Proxy =
            UnsafeUpgrades.deployUUPSProxy(collateralPoolV2Impl,
            ↪  abi.encodeCall(CollateralPoolV2.initialize, (owner)));
        collateralPoolV2 = CollateralPoolV2(payable(collateralPoolV2Proxy));
        collateralPoolV2.setFactory(address(factory));

        factory.setCollateralPoolV2(address(collateralPoolV2));
        // Register owner in collateralPoolV2 and deposit funds
        collateralPoolV2.setWhitelistedToken(address(baseToken), true);
        collateralPoolV2.setWhitelistedToken(address(quoteToken), true);

        // Give owner and users some ETH for gas
        vm.deal(owner, 10 ether);
        vm.deal(alice, 10 ether);
        vm.deal(bob, 10 ether);

        // Give users some USDC for testing
        deal(address(quoteToken), alice, USER_INIT_WEALTH_USDC);
        deal(address(quoteToken), bob, USER_INIT_WEALTH_USDC);

        setUpLabels();
    }

    // Helper functions
    function _createVault(bool _isBuyLow) private {
        baseToken.approve(address(collateralPoolV2), type(uint256).max);
        quoteToken.approve(address(collateralPoolV2), type(uint256).max);

        collateralPoolV2.deposit(address(baseToken), OWNER_INIT_WEALTH_AERO);
        collateralPoolV2.deposit(address(quoteToken), 1.52e7);

        CreateVaultParams memory vaultParams = CreateVaultParams({
            owner: owner,
            baseToken: address(baseToken),
            quoteToken: address(quoteToken),
            expiry: vaultTestData.expiry,
            linkedOraclePrice: vaultTestData.linkedOraclePrice,
            yieldValue: vaultTestData.yieldValue,
            isBuyLow: _isBuyLow,
            quantity: _isBuyLow ? vaultTestData.quantityBuyLow :
             ↪  vaultTestData.quantitySellHigh,
            useCollateralPool: false,
            useNativeToken: false,
            vaultSeriesVersion: 2
        });

        vaultV2_1 = VaultV2(payable(factory.createVault{value: 10}(vaultParams, new
         ↪  bytes[](0x1))));
        collateralPoolV2.approveVault(address(vaultV2_1), true);
    }

    function _createETHVault() private {
        collateralPoolV2.setWhitelistedToken(address(WETHToken), true);
        oraclePriceAtCreation = 2500e6; // 1 WETH = 2500 USDC
        mockPythOracle.setCurrentPrice(2500e8);
        factory.setTokenPairAggregator(WETHToken, USDCToken,
         ↪  address(pythAggregator), wethUsdcTestData.priceFeed);
```

```solidity
        factory.setWETH(WETHToken);

        IERC20(address(WETHToken)).approve(address(collateralPoolV2),
        ↪   type(uint256).max);
        IERC20(address(USDCToken)).approve(address(collateralPoolV2),
        ↪   type(uint256).max);

        collateralPoolV2.deposit(address(WETHToken), OWNER_INIT_WEALTH_WETH);
        collateralPoolV2.deposit(address(USDCToken), OWNER_INIT_WEALTH_USDC);

        CreateVaultParams memory vaultParams = CreateVaultParams({
            owner: owner,
            baseToken: address(WETHToken),
            quoteToken: address(USDCToken),
            expiry: wethUsdcTestData.expiry,
            linkedOraclePrice: wethUsdcTestData.linkedOraclePrice,
            yieldValue: wethUsdcTestData.yieldValue,
            isBuyLow: false,
            quantity: wethUsdcTestData.quantitySellHigh,
            useCollateralPool: false,
            useNativeToken: false,
            vaultSeriesVersion: 2
        });

        vaultV2_1 = VaultV2(payable(factory.createVault{value: 10}(vaultParams, new
        ↪   bytes[](0x1))));
        collateralPoolV2.approveVault(address(vaultV2_1), true);
    }

    function _depositForUser(address user, uint256 amount) private {
        vm.startPrank(user);
        address token = vaultV2_1.investmentToken();
        IERC20(token).approve(address(vaultV2_1), amount);
        vaultV2_1.deposit{value: 10}(user, amount, new bytes[](0x1));
        vm.stopPrank();
    }

    function _depositForUserWithSignature(address user, uint256 amount, uint256
    ↪   yieldValue, bytes memory signature)
        private
    {
        vm.startPrank(user);
        address token = vaultV2_1.investmentToken();
        IERC20(token).approve(address(vaultV2_1), amount);
        vaultV2_1.deposit{value: 10}(user, amount, yieldValue, nonce, deadline,
        ↪   signature, new bytes[](0x1));
        vm.stopPrank();
    }
```

```
function _setupDepositSignature(address user, uint256 amount, uint256
 ↪ yieldValue) private returns (bytes memory) {
    // Change signer
    vaultV2_1.setSigner(signer);

    bytes32 messageHash = keccak256(abi.encodePacked(address(vaultV2_1), user,
     ↪ amount, yieldValue, nonce, deadline));
    bytes32 ethSignedMessageHash =
     ↪ MessageHashUtils.toEthSignedMessageHash(messageHash);
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(signerPrivateKey,
     ↪ ethSignedMessageHash);
    bytes memory validSignature = abi.encodePacked(r, s, v);

    return validSignature;
}

function _calculatePositionAmounts(uint256 principal, bool isBuyLow)
    private
    view
    returns (uint256 linkedTokenTotal, uint256 investTokenTotal, uint256
     ↪ tradingFee)
{
    uint256 yieldValue = vaultTestData.yieldValue;
    uint256 linkedPrice = vaultTestData.linkedPrice;

    (linkedTokenTotal, investTokenTotal) =
        CalcLib.calculatePositionAmounts(principal, yieldValue, isBuyLow,
         ↪ linkedPrice);
    tradingFee = CalcLib.calculateTradingFee(principal, yieldValue, isBuyLow,
     ↪ tradingFeeRate, oraclePriceAtCreation);
}

function _getLpBalance(address _owner, address token) internal view returns
 ↪ (LpBalance memory) {
    (uint256 ownDeposit, uint256 reservedForVaults, uint256 usersTotalDeposit,
     ↪ uint256 borrowedFromUsers) =
        collateralPoolV2.lpBalance(_owner, token);

    LpBalance memory balance;
    balance.ownDeposit = ownDeposit;
    balance.reservedForVaults = reservedForVaults;
    balance.usersTotalDeposit = usersTotalDeposit;
    balance.borrowedFromUsers = borrowedFromUsers;
    // ______gap is automatically initialized to zeros

    return balance;
}


function test_Shortfall_POC() public {
```

```
// ----- Condition -----
_createVault(true);
address VaultOwnerAndLP = vaultV2_1.owner();
console.log("Vault owner:", VaultOwnerAndLP);
console.log("CollateralPool owner:", collateralPoolV2.owner());
console.log("Is whitelisted:",
↪  collateralPoolV2.whitelistedLP(VaultOwnerAndLP));

vm.prank(collateralPoolV2.owner());
collateralPoolV2.setWhitelistedLP(VaultOwnerAndLP, true);
console.log("Is whitelisted:",
↪  collateralPoolV2.whitelistedLP(VaultOwnerAndLP));
_depositForUser(alice, aliceDepositUSDC);

(uint256 usdcOwnDeposit, uint256 usdcReserved, uint256 usdcUsersTotal,
↪  uint256 usdcBorrowed) = collateralPoolV2.lpBalance(VaultOwnerAndLP,
↪  address(quoteToken));
console.log("LP USDC ownDeposit: %e", usdcOwnDeposit, "borrowedFromUsers:
↪  %e", usdcBorrowed);

(uint256 alicePrincipal, uint256 aliceLinkedTotal, uint256 aliceYield) =
↪  collateralPoolV2.userVaultBalance(address(vaultV2_1), alice);
console.log("Alice principal: %e", alicePrincipal);
console.log("Alice yield: %e", aliceYield);

(, uint256 aliceInvestmentTokenYield,) =
↪  _calculatePositionAmounts(aliceDepositUSDC, true);

// Warp to expiry
vm.warp(vaultTestData.expiry + 2 hours);

// Set oracle price above linked price (swap condition NOT met for buyLow)
mockPythOracle.setCurrentPrice(int64(1.5e8)); // Price above 0.5

(uint256 usdcOwnDeposit2, uint256 usdcReserved2, uint256 usdcUsersTotal2,
↪  uint256 usdcBorrowed2) = collateralPoolV2.lpBalance(VaultOwnerAndLP,
↪  address(quoteToken));
console.log("LP USDC ownDeposit: %e", usdcOwnDeposit2, "borrowedFromUsers:
↪  %e", usdcBorrowed2);

        // ----- Action -----
// Execute
uint8 state = vaultV2_1.execute{value: 10}(new bytes[](0x1),
↪  getPriceOptions);
assertEq(state, 2); // Unswapped

console.log("Execute Worked");

uint256 shortfall = collateralPoolV2.previewShortfall(VaultOwnerAndLP,
↪  address(quoteToken));
```

```
        console.log("Shortfall: %e", shortfall);

        vm.prank(alice);
        vm.expectRevert(abi.encodePacked("CollateralPoolV2: insufficient
        ↪  investmentToken balance for vault"));
        vaultV2_1.withdraw(new bytes[](0x1), getPriceOptions);
    }

}
```

Run the test with the command `forge test --mt test_Shortfall_POC -vv --via-ir`

Console Output:

```
Logs:
  Vault owner: 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496
  CollateralPool owner: 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496
  Is whitelisted: false
  Is whitelisted: true
  LP USDC ownDeposit: 1.49e7 borrowedFromUsers: %e 0
  Alice principal: 2.5e8
  Alice yield: 1.5e7
  LP USDC ownDeposit: 1.49e7 borrowedFromUsers: %e 0
  Execute Worked
  Shortfall: 1e5

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 6.39s (655.46ms CPU
 ↪  time)
```

Tool Used

Manual Review

Recommendation

Consider adding support for partial withdrawals so users can withdraw the available portion while tracking any remaining owed amount.

Discussion

shawnwang0715

Thanks for pointing this out. Since we are the only whitelisted LP, we can guarantee that we will have enough tokens to fill the shortfall. We won't need to fix this issue now.

Lucas | Sherlock

Issue acknowledged by the team. Won't be fixed at this time.

Issue M-2: Owner can front-run depositors by calling `adjustYieldValue` and setting it to dust amounts [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-09-prodigy-finance-sept-25th/issues/14

Summary

Owners front-run depositors by setting the `yieldValue` to dust amounts, so users get no yield and no trading fee is enacted on the owner for the deposit as well.

Vulnerability Detail

The owner can front-run any depositor by setting `yieldValue` to a dust amount, this way the depositor will not get any yield after he deposits and also no trading fee will be enacted on the owner.

He can then back-run the deposit, setting the value back to the original one. The contract are planned to be deployed on mainnet where MEV style attacks are easily doable. While users may lose confidence in the owner, they have to wait for the option to finish in order to get their funds back.

Impact

Frozen funds, as users will deposit for the yield, this way they are just freezing their funds in the contract until `VaultCore::execute` is called and the Vault is closed.

Code Snippet

https://github.com/sherlock-audit/2025-09-prodigy-finance-sept-25th/blob/3ce9fb38cffe4a26e601e2d556b56b201a2a513b/brt-dci-contracts/src/Vault.sol#L145

Tool Used

Manual Review

Recommendation

We recommend adding a `minYieldValue` for depositors, so they can choose what minimum yield they would accept when depositing. Another fix can be adding a timelock to `adjustYieldValue`, where changes to `yieldValue` take effect after X number of blocks/seconds/hours etc.

Issue L-1: Whitelisted LPs bypass balance checks, exposing users to withdrawal shortfall risk [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-09-prodigy-finance-sept-25th/issues/11

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Whitelisted LPs bypass balance checks during deposits

Vulnerability Detail

When an LP is whitelisted, the `processUserDeposit` function skips the balance verification checks, and assumes the LP will later cover any shortfall before withdrawals post-expiry.

Impact

If the whitelisted LP is unable to cover a shortfall, users who deposited will be unable to withdraw their balances.

Code Snippet

https://github.com/sherlock-audit/2025-09-prodigy-finance-sept-25th/blob/3ce9fb38cffe4a26e601e2d556b56b201a2a513b/brt-dci-contracts/src/CollateralPoolV2.sol#L177-L184

Tool Used

Manual Review

Recommendation

Consider always performing balance checks, even for whitelisted LPs.

Otherwise, consider documenting this behaviour so depositors are aware of the trust assumptions associated with whitelisted lp's/vault owners.

Discussion

shawnwang0715

Thanks for pointing this out. Since we are the only whitelisted LP, we can guarantee that we will have enough tokens to fill the shortfall. We won't need to fix this issue now.

Lucas | Sherlock

Issue acknowledged by the team. Won't be fixed at this time.

Issue L-2: `_transferVaultCreationFunds` has a leaky check for `requiredETH` [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-09-prodigy-finance-sept-25th/issues/13

Description Using `msg.value` here is not correct 100% of the time, in the case where the aggregator.getPrice uses getLatestEmaPrice, the `PythAggregator` will attempt to updatePriceFeeds and refund any excess back to the Factory.

When we get to `msg.value < requiredETH`, `msg.value` no longer represents the correct value, as some of `msg.value` was used for Pyth, thus the check can pass when it shouldn't.

Impact No security impact, as if there aren't enough funds in the contract `weth.deposit` will revert, but the check is technically wrong

Line ref https://github.com/sherlock-audit/2025-09-prodigy-finance-sept-25th/blob/3ce9fb38cffe4a26e601e2d556b56b201a2a513b/brt-dci-contracts/src/Factory.sol#L262

Recommendation Use `address(this).balance` instead or cache the original `msg.value` and decrement it if Pyth used some of it

Issue L-3: Missing chain ID in signed message hash enables cross-chain replay [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-09-prodigy-finance-sept-25th/issues/15

Summary

The message hash for signed deposits in VaultV2 is calculated in the following way:

```
bytes32 messageHash =
    keccak256(abi.encodePacked(address(this), to, depositAmount, _yieldValue,
     ↪  nonce, deadline));
bytes32 ethSignedMessageHash = MessageHashUtils.toEthSignedMessageHash(messageHash);
```

Since this does not include the chainId in the signature, the signatures can be re-used across chains, as long as the vault is deployed at the same address (and the expected signer is the same).

Vulnerability Detail

Impact

In certain cases, signed deposits can be made on chains where the signature was not intended to be used. This allows the `yieldValue` to be changed to a potentially unintended value.

Code Snippet

https://github.com/sherlock-audit/2025-09-prodigy-finance-sept-25th/blob/3ce9fb38cffe4a26e601e2d556b56b201a2a513b/brt-dci-contracts/src/VaultV2.sol#L102-L105

Tool Used

Manual Review

Recommendation

Consider including `block.chainid` in the message to be signed

Disclaimers