// Smart Contract Security Assessment 10.10.2024 - 10.14.2024

Brt Dci Contracts

Prodigy

HALBORN

Brt Dci Contracts - Prodigy

Prepared by: HALBORN

Last Updated 10/30/2024

Date of Engagement by: October 10th, 2024 - October 14th, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
11	0	1	4	4	2

TABLE OF CONTENTS

1. Introduction

`Prodigy` engaged Halborn to conduct a security assessment on their smart contracts revisions beginning on *10/10/2024* and ending on *10/14/2024*. The security assessment was scoped to the smart contracts provided to the Halborn team.

2. Assessment Summary

The team at Halborn was provided 3 days for the engagement and assigned a full-time security engineer to evaluate the security of the smart contract.

The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some security risks that were addressed by the `Prodigy team`.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance code coverage and quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (`solgraph,draw.io`)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Testnet deployment. (`Hardhat`,`Foundry`)

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: brt-dci-contracts

(b) Assessed Commit ID: 24946ea

(c) Items in scope:

- DCIStructs
- TransferHelper
- AggregatorHelper
- Factory
- PythAggregator
- Vault
- VaultKeeper
- Router

Out-of-Scope: utils/Faucet, utils/Token

REMEDIATION COMMIT ID:

- d422f6c
- ef55a35

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	4	4	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
PRICE MANIPULATION VULNERABILITY IN VAULT EXECUTION DUE TO UNCHECKED PYTH ORACLE UPDATES	HIGH	SOLVED - 10/29/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
CHAINLINK ORACLE PRICE FEED USED WITHOUT STALENESS CHECK	MEDIUM	SOLVED - 10/29/2024
EXCESS ETH NOT REFUNDED IN PRICE UPDATE TRANSACTIONS	MEDIUM	SOLVED - 10/29/2024
ZERO AMOUNT TRANSFER VULNERABILITY IN TOKEN TRANSFERS	MEDIUM	SOLVED - 10/29/2024
PYTH ORACLE PRICE IS NOT VALIDATED PROPERLY	MEDIUM	SOLVED - 10/29/2024
UNRESTRICTED VAULT CREATION IN FACTORY CONTRACT	LOW	SOLVED - 10/29/2024
UNSAFE CASTING OPERATIONS	LOW	SOLVED - 10/29/2024
INCORRECT FEE CALCULATION DUE TO DELAYED INITIALIZATION IN VAULT CONTRACT	LOW	SOLVED - 10/29/2024
INCORRECT STATE MODIFICATION ORDER IN LPWITHDRAW FUNCTION	LOW	SOLVED - 10/29/2024
MISSING VISIBILITY ATTRIBUTE	INFORMATIONAL	SOLVED - 10/29/2024
CONSIDER USING NAMED MAPPINGS	INFORMATIONAL	SOLVED - 10/29/2024

7. FINDINGS & TECH DETAILS

7.1 PRICE MANIPULATION VULNERABILITY IN VAULT EXECUTION DUE TO UNCHECKED PYTH ORACLE UPDATES

// HIGH

Description

The `execute` function in Vault.sol allows arbitrary `priceUpdateData` to be passed, which is used to fetch the oracle price:

```
function execute(bytes[] calldata priceUpdateData) public payable initialized returns
(uint8) {
    // ...
    int256 oraclePriceAtExpiry = aggregator.getPrice(priceFeed, priceUpdateData);
    // ...
}
```

This `priceUpdateData` is passed through `AggregatorHelper.sol` and then to `PythAggregator.sol` :

```
function getLatestEmaPrice(bytes32 _pythPriceFeed, bytes[] calldata _priceUpdateData)
external payable returns (PythStructs.Price memory) {
    updatePrice(_priceUpdateData);
    return getEmaPrice(_pythPriceFeed);
}

function updatePrice(bytes[] calldata _priceUpdateData) public payable {
    uint256 fee = pyth.getUpdateFee(_priceUpdateData);
    pyth.updatePriceFeeds{value: fee}(_priceUpdateData);
}
```

The issue is that there's no validation of `_priceUpdateData`. A malicious user can provide empty price data, causing the `updatePrice` function to skip the update entirely.
A vault executor could then skip or avoid the updating of the Pyth price by passing in an empty `priceUpdateData` array.
Link to Base Pyth UpdatePriceFeeds

```
function updatePriceFeeds(
    bytes[] calldata updateData
) public payable override {
    uint totalNumUpdates = 0;
    for (uint i = 0; i < updateData.length; ) {
        if (
            updateData[i].length > 4 &&
            UnsafeCalldataBytesLib.toUint32(updateData[i], 0) ==
```

```
            ACCUMULATOR_MAGIC
        ) {
            totalNumUpdates += updatePriceInfosFromAccumulatorUpdate(
                updateData[i]
            );
        } else {
            updatePriceBatchFromVm(updateData[i]);
            totalNumUpdates += 1;
        }

        unchecked {
            i++;
        }
    }
    uint requiredFee = getTotalFee(totalNumUpdates);
    if (msg.value < requiredFee) revert PythErrors.InsufficientFee();
}
```

This leads to the `getEmaPrice` function potentially returning an outdated price, which is then used to determine the vault's state.
Ref 1
Ref 2

Proof of Concept

Actors:

- Alice: An honest user who created a vault
- Bob: A malicious user looking to exploit the system

Initial Setup:
1. Alice creates a "Buy Low" vault with the following parameters:

- Linked Price: 1000 USDC per ETH
- Expiry: 7 days from now

2. The current market price of ETH is 1100 USDC.
Exploit Scenario:
1. 7 days pass, and the vault is ready for execution.
2. The current market price of ETH has dropped to 990 USDC.
3. Bob, being a malicious actor, notices this price drop and sees an opportunity to exploit the system.
4. Bob calls the `execute` function on Alice's vault, but instead of providing the current price data, he provides an empty `priceUpdateData` array.
5. The `execute` function passes this empty array through the system:

- Vault.sol calls AggregatorHelper.sol
- AggregatorHelper.sol calls PythAggregator.sol
- PythAggregator.sol's `updatePrice` function doesn't perform any update due to the empty array
- PythAggregator.sol's `getEmaPrice` function returns the last stored price, which is 1010 USDC

6. The `execute` function uses this outdated price of 1010 USDC to determine the vault's state.
7. Since 1010 USDC is greater than the Linked Price of 1000 USDC, the vault's state is set to "unswapped" (state = 2).
8. As a result, Alice's "Buy Low" strategy fails to execute, even though the actual market price (990 USDC) is below her Linked Price.
Outcome:

- Alice loses the opportunity to buy ETH at her desired price, even though market conditions were favorable.
- Bob potentially benefits if he's an LP or has opposing positions.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:M/Y:M/R:N/S:C (8.6)

Recommendation

It is recommended to implement a strict validation of the `priceUpdateData` in the `execute` function of Vault.sol, mostly when `execute()` is called with Pyth Oracle:

```
require(priceUpdateData.length > 0, "Price update data cannot be empty");
```

A modifier can also be implemented to always verify this.

Remediation

SOLVED: A modifier has been implemented to handle 0 length `priceUpdateData` array :

```
modifier checkPriceUpdateData(bytes[] calldata _priceUpdateData) {
        require(_priceUpdateData.length > 0, "PythAggregator: priceUpdateData is
empty");
        _;
    }
```

Remediation Hash

https://github.com/prodigyfi/brt-dci-contracts/commit/d422f6cefa174580ce3afe6c03570cff8551149b

References

prodigyfi/brt-dci-contracts/src/Vault.sol#L121

7.2 CHAINLINK ORACLE PRICE FEED USED WITHOUT STALENESS CHECK

// MEDIUM

Description

The `getPrice` function in AggregatorHelper.sol fetches price data from a Chainlink oracle without validating the freshness of the returned data:

```
function getPrice(address _aggregator, bytes32 _pythPriceFeed, bytes[] calldata
_priceUpdateData)
    internal
    returns (int256 price)
{
    if (_pythPriceFeed == bytes32(0)) {
// latestRoundData() returns int256 answer
        (, price,,,) = IChainlinkOracle(_aggregator).latestRoundData();
    } else {
// PythStructs.Price.price is int64
        price =
            IPythAggregator(_aggregator).getLatestEmaPrice{value: msg.value}
(_pythPriceFeed, _priceUpdateData).price;
    }
}
```

The function calls `latestRoundData()` on the Chainlink oracle but does not check the `updatedAt` timestamp to ensure the price data is current. This allows stale or outdated price information to be used in critical calculations.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:M/R:N/S:C (6.3)

Recommendation

It is recommended to implement a staleness check using the `updatedAt` value returned by `latestRoundData()`. Compare it against the current block timestamp and a predefined threshold:

```
if (_pythPriceFeed == bytes32(0)) {
        (, price,,uint256 updatedAt,) =
IChainlinkOracle(_aggregator).latestRoundData();
        require(block.timestamp - updatedAt <= PRICE_FRESHNESS_THRESHOLD, "Stale
price data");
    }
```

Remediation

SOLVED: Checks have been implemented to ensure price returned by chainlink is correct and not stale.

Remediation Hash

https://github.com/prodigyfi/brt-dci-contracts/commit/d422f6cefa174580ce3afe6c03570cff8551149b

References

prodigyfi/brt-dci-contracts/src/AggregatorHelper.sol#L30

7.3 EXCESS ETH NOT REFUNDED IN PRICE UPDATE TRANSACTIONS

// MEDIUM

Description

The `PythAggregator` contract's `updatePrice` function, which is called indirectly through `getPrice()`, does not refund excess ETH sent by users. This function is called in multiple payable functions across `Router.sol`, `Factory.sol`, and `Vault.sol`, allowing users to provide `msg.value`. The relevant code in `PythAggregator.sol` is:

```
function updatePrice(bytes[] calldata _priceUpdateData) public payable {
    uint256 fee = pyth.getUpdateFee(_priceUpdateData);
    pyth.updatePriceFeeds{value: fee}(_priceUpdateData);
}
```

This implementation transfers the exact fee to the `Pyth` contract but does not refund any excess ETH sent by the user. The excess ETH remains trapped in the `PythAggregator` contract.
Users who inadvertently send more ETH than required for price updates will lose their excess funds. This results in a direct financial loss for users and accumulation of unusable ETH in the `PythAggregator` contract. The severity of this issue is heightened by the fact that it affects multiple core functions of the protocol in Router.sol,Factory.sol and of course Vault.sol

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:N/R:N/S:C (6.3)

Recommendation

It is recommended to implement a refund mechanism in the `updatePrice` function to return any excess ETH to the user. Here's an example of how to modify the function:

```
function updatePrice(bytes[] calldata _priceUpdateData) public payable {
    uint256 fee = pyth.getUpdateFee(_priceUpdateData);
    pyth.updatePriceFeeds{value: fee}(_priceUpdateData);
    uint256 excess = msg.value - fee;
    if (excess > 0) {
        (bool success, ) = msg.sender.call{value: excess}("");
        require(success, "ETH refund failed");
    }
}
```

Remediation

SOLVED: Excess fees are now refunded to `msg.sender`.

Remediation Hash

https://github.com/prodigyfi/brt-dci-contracts/commit/d422f6cefa174580ce3afe6c03570cff8551149b

References

prodigyfi/brt-dci-contracts/src/PythAggregator.sol#L42

7.4 ZERO AMOUNT TRANSFER VULNERABILITY IN TOKEN TRANSFERS

// MEDIUM

Description

The contracts contain multiple instances call of `safeTransfer()` and `safeTransferFrom()` calls that do not check if the transfer amount is greater than zero. This issue is present in the following locations:

- In the init() function:

```
linkedToken.safeTransferFrom(owner(), address(this), linkedTokenAmount);
investmentToken.safeTransferFrom(owner(), address(this), investmentTokenAmount +
fees);
```

- In the deposit() function:

```
investmentToken.safeTransferFrom(msg.sender, address(this), depositAmount);
```

- In the withdraw() function:

```
linkedToken.safeTransfer(msg.sender, linkedTokenAmount);
investmentToken.safeTransfer(msg.sender, investmentTokenAmount);
```

- In the _lpWithdrawInvestmentToken() function:

```
investmentToken.safeTransfer(msg.sender, investmentTokenAmount);
```

- In the _lpWithdrawLinkedToken() function:

```
linkedToken.safeTransfer(msg.sender, linkedTokenAmount);
```

- In the lpWithdraw() function:

```
investmentToken.safeTransfer(feeReceiver, fees);
linkedToken.safeTransfer(feeReceiver, fees);
investmentToken.safeTransfer(msg.sender, investmentTokenAmount);
linkedToken.safeTransfer(msg.sender, linkedTokenAmount);
```

- In the lpCancel() function:

```
investmentToken.safeTransferFrom(msg.sender, feeReceiver, cancellationFee);
linkedToken.safeTransferFrom(msg.sender, feeReceiver, cancellationFee);
```

These transfers are executed without verifying if the amount is greater than zero. Some ERC20 tokens, such as older implementations or non-standard tokens, revert when a transfer amount of zero is attempted. The absence of a zero amount check can cause these transfers to fail and revert the entire transaction.

Refs :

- Solodit-Caviar
- Solodit-Concur

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:C (6.3)

Recommendation

To address this, it is recommended to implement zero-amount checks before all token transfers. Add a condition to skip the transfer if the amount is zero. Here's an example of how to modify the code:

```
if (amount > 0) {
    token.safeTransfer(receiver, amount);
}
```

Remediation

SOLVED: Checks for non 0 amounts have been added to the needed functions.

Remediation Hash

https://github.com/prodigyfi/brt-dci-contracts/commit/d422f6cefa174580ce3afe6c03570cff8551149b

References

prodigyfi/brt-dci-contracts/src/Vault.sol#L169

7.5 PYTH ORACLE PRICE IS NOT VALIDATED PROPERLY

// MEDIUM

Description

The `PythAggregator` contract does not perform input validation on the `price`, `conf`, and `expo` values returned from the called price feed, which can lead to the contract accepting invalid or untrusted prices. Those values should be checked as clearly stated in the official documentation.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:L/Y:N/R:N/S:C (4.7)

Recommendation

It is recommended that the contract revert the transaction here if one of the following conditions is triggered:

- `price <= 0`
- `expo < -18`
- `conf > 0 && (price / int64(conf) < MIN_CONF_RATIO` for a given `MIN_CONF_RATIO`

Remediation

SOLVED: Pyth price now have checks about `price`,`expo` and `conf.`

Remediation Hash

https://github.com/prodigyfi/brt-dci-contracts/commit/ef55a358607299c68249a1705895e524546a8a56

References

https://solodit.xyz/issues/m-01-pyth-oracle-price-is-not-validated-properly-pashov-audit-group-none-nabla-markdown

7.6 UNRESTRICTED VAULT CREATION IN FACTORY CONTRACT

// LOW

Description

The Factory.sol contract allows unrestricted vault creation, contradicting the documented requirement for Liquidity Providers (LPs). The createVault() function lacks access control:

```
function createVault(DCIStructs.CreateVaultParams calldata _createVaultParams,
bytes[] calldata _priceUpdateData)
    external
    payable
    whenNotPaused
    returns (address newVault)
{
// Function implementation
}
```

This implementation permits any address to create vaults without authorization checks.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:C (3.1)

Recommendation

It is recommended to either modify the documentation or implement strict access control for vault creation.

Remediation

SOLVED: Documentation has been updated.

Remediation Hash

https://github.com/prodigyfi/brt-dci-contracts/commit/ef55a358607299c68249a1705895e524546a8a56

References

prodigyfi/brt-dci-contracts/src/Factory.sol#L178

7.7 UNSAFE CASTING OPERATIONS

// LOW

Description

In the AggregatorHelper.sol contract, several unsafe casting operations are performed without using `SafeCast` library. This could potentially lead to unexpected overflows or underflows in certain edge cases.
The following unsafe casting operations were identified:

1. `int32(-1 * int8(IChainlinkOracle(_aggregator).decimals()))`
2. `uint256(uint64(price)) * 10 ** uint32(targetDecimals - priceDecimals)`
3. `uint256(uint64(price)) / 10 ** uint32(priceDecimals - targetDecimals)`

These operations involve casting between different integer types (`int8`, `int32`, `uint64`, `uint256`) without using SafeCast library to prevent potential overflows or underflows.
Unsafe casting could potentially lead to unexpected behavior or vulnerabilities in edge cases or future modifications of the contract.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:C (3.1)

Recommendation

It is recommended to consider using OpenZeppelin's `SafeCast` library for all casting operations to prevent potential overflows or underflows. This is a best practice that enhances the overall robustness and security of the contract.

Remediation

SOLVED: SafeCast is now used everywhere.

Remediation Hash

https://github.com/prodigyfi/brt-dci-contracts/commit/d422f6cefa174580ce3afe6c03570cff8551149b

References

prodigyfi/brt-dci-contracts/src/AggregatorHelper.sol#L57

7.8 INCORRECT FEE CALCULATION DUE TO DELAYED INITIALIZATION IN VAULT CONTRACT

// LOW

Description

The Vault contract calculates fees during the `init()` function call instead of at contract creation. This delay allows for a discrepancy between the price at contract creation and the price at initialization.

Relevant code snippets:

```
// In constructor
constructor(DCIStructs.VaultParams memory _vaultParams, DCIStructs.FeeParams memory
_feeParams)
    Ownable(_vaultParams.owner)
{
    // ... other initializations ...
    oraclePriceAtCreation = _feeParams.oraclePriceAtCreation;
}

// In init() function
function init() external onlyOwner {
    require(!isInitialized, "Vault: contract instance has already been initialized");
    require(block.timestamp <= depositDeadline, "Vault: deposit period has ended");

    // ... other logic ...

    uint256 fees = _calculateFees(quantity);

    // ... fee transfer logic ...

    isInitialized = true;
    emit Initialize();
}

// Fee calculation function
function _calculateFees(uint256 amount) internal view returns (uint256 fees) {
    if (isBuyLow) {
        fees = amount * listingFeeRate / 1e18;
    } else {
        fees = amount * listingFeeRate * SafeCast.toUint256(oraclePriceAtCreation) /
1e36;
    }
}
```

This leads to incorrect fee calculations. A malicious vault creator could exploit this by:
1. Deploying the contract when prices are high.
2. Waiting for prices to drop.
3. Calling `init()` when prices are low, resulting in lower fees.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:L/Y:N/R:N/S:C (3.1)

Recommendation

It is recommended to calculate and store the fees in the constructor to ensure they are based on the price at contract creation.

Remediation

SOLVED: Fees are still computed in constructor but added to `linkedTokenAmount` or `investmentTokenAmount` so the problem is solved.

Remediation Hash

https://github.com/prodigyfi/brt-dci-contracts/commit/d422f6cefa174580ce3afe6c03570cff8551149b

References

prodigyfi/brt-dci-contracts/src/Vault.sol#L79

7.9 INCORRECT STATE MODIFICATION ORDER IN LPWITHDRAW FUNCTION

// LOW

Description

The `lpWithdraw` function in the Vault contract violates the Checks-Effects-Interactions (CEI) pattern. The `lpWithdrawn` state variable is set to true after external interactions are performed, allowing for potential reentrancy attacks. The problematic code is as follows:

```
function lpWithdraw(bytes[] calldata priceUpdateData) external payable onlyOwner
initialized {
    if (state == 0) execute(priceUpdateData);
    require(block.timestamp >= withdrawDate, "Vault: withdraw date has not passed
yet");
    require(!lpWithdrawn, "Vault: LP has withdrawn already");

    uint256 linkedTokenAmount;
    uint256 investmentTokenAmount;

    if (depositTotal != 0) {
        uint256 fees = _calculateFees(depositTotal);
        (isBuyLow ? investmentToken : linkedToken).safeTransfer(feeReceiver, fees);
    }

    if (state == 1) {
        investmentTokenAmount = IERC20(investmentToken).balanceOf(address(this));
        investmentToken.safeTransfer(msg.sender, investmentTokenAmount);
        _lpWithdrawLinkedToken(false);
    } else {
        linkedTokenAmount = IERC20(linkedToken).balanceOf(address(this));
        linkedToken.safeTransfer(msg.sender, linkedTokenAmount);
        _lpWithdrawInvestmentToken(false);
    }

    //E @audit CEI NOT RESPECTED
    lpWithdrawn = true;

    emit LpWithdraw(linkedTokenAmount, investmentTokenAmount);
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:C (3.1)

Recommendation

To mitigate this, it is recommended to implement the Checks-Effects-Interactions pattern by updating the `lpWithdrawn` state variable before performing any external calls. Restructure the function as follows:

```
function lpWithdraw(bytes[] calldata priceUpdateData) external payable onlyOwner
initialized {
    if (state == 0) execute(priceUpdateData);
    require(block.timestamp >= withdrawDate, "Vault: withdraw date has not passed
yet");
    require(!lpWithdrawn, "Vault: LP has withdrawn already");

    lpWithdrawn = true;

    // ... //

    emit LpWithdraw(linkedTokenAmount, investmentTokenAmount);
}
```

Remediation

SOLVED: `lpWithdrawn = true;` is now set right after the require statement.

Remediation Hash

https://github.com/prodigyfi/brt-dci-contracts/commit/d422f6cefa174580ce3afe6c03570cff8551149b

References

prodigyfi/brt-dci-contracts/src/Vault.sol#L216

7.10 MISSING VISIBILITY ATTRIBUTE

// INFORMATIONAL

Description

It is best practice to set the visibility of state variables and constants explicitly.

```
bool hasEarlyWithdrawn = false;
bool lpWithdrawn = false;
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

Consider explicitly setting the visibility of all state variables and constants.

Remediation

SOLVED: Each variable has now a visibility set.

Remediation Hash

https://github.com/prodigyfi/brt-dci-contracts/commit/d422f6cefa174580ce3afe6c03570cff8551149b

References

prodigyfi/brt-dci-contracts/src/Vault.sol#L36C5-L37C30

7.11 CONSIDER USING NAMED MAPPINGS

// INFORMATIONAL

Description

Using named mappings can improve the readability and maintainability of the code by making the purpose of each mapping clearer. This practice helps developers and auditors understand the mappings' intent more easily.

```
mapping(bytes32 => uint8) public tokenPairDecimals;
mapping(address => uint256) public balances;
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

Consider refactoring the mappings to use named arguments, which will enhance code readability and make the purpose of each mapping more explicit.

```
mapping(bytes32 tokenPair => uint8 decimals) public tokenPairDecimals;
mapping(address user => uint256 balance) public balances;
```

Remediation

SOLVED: Mappings are now using named arguments.

Remediation Hash

https://github.com/prodigyfi/brt-dci-contracts/commit/d422f6cefa174580ce3afe6c03570cff8551149b

References

prodigyfi/brt-dci-contracts/src/Vault.sol#L41

8. AUTOMATED TESTING

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework.
After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

```
→  brt-dci-contracts git:(master) ✗ slither . --exclude-low --exclude-informational
'forge clean' running (wd: /Users/liliancariou/Desktop/Halborn/audits/brt-dci-contracts)
'forge config --json' running
'forge build --build-info --skip */test/** */script/** --force' running (wd: /Users/liliancariou/Desktop/Halborn/audits/brt-dci-contracts)
INFO:Detectors:
Vault.init() (src/Vault.sol#102-131) uses arbitrary from in transferFrom: linkedToken.safeTransferFrom(owner(),address(this),linkedTokenAmount) (src/Vault.sol#121)
Vault.init() (src/Vault.sol#102-131) uses arbitrary from in transferFrom: investmentToken.safeTransferFrom(owner(),address(this),investmentTokenAmount + fees) (src/Vault.sol#122)
Vault.init() (src/Vault.sol#102-131) uses arbitrary from in transferFrom: linkedToken.safeTransferFrom(owner(),address(this),linkedTokenAmount + fees) (src/Vault.sol#125)
Vault.init() (src/Vault.sol#102-131) uses arbitrary from in transferFrom: investmentToken.safeTransferFrom(owner(),address(this),investmentTokenAmount) (src/Vault.sol#126)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
PythAggregator.updatePrice(bytes[]) (src/PythAggregator.sol#57-61) sends eth to arbitrary user
        Dangerous calls:
        - pyth.updatePriceFeeds{value: fee}(_priceUpdateData) (src/PythAggregator.sol#60)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
Reentrancy in Vault.lpWithdraw(bytes[]) (src/Vault.sol#182-222):
        External calls:
        - execute(priceUpdateData) (src/Vault.sol#185)
                - oraclePriceAtExpiry = aggregator.getPrice(priceFeed,priceUpdateData) (src/Vault.sol#298)
                - price = IPythAggregator(_aggregator).getLatestEmaPrice{value: msg.value}(_pythPriceFeed,_priceUpdateData).price (src/AggregatorHelper.sol#33)
        - investmentToken.safeTransfer(msg.sender,investmentTokenAmount) (src/Vault.sol#211)
        - _lpWithdrawLinkedToken(false) (src/Vault.sol#212)
                - (success,data) = token.call(abi.encodeWithSelector(0xa9059cbb,to,value)) (src/TransferHelper.sol#15)
                - linkedToken.safeTransfer(msg.sender,linkedTokenAmount) (src/Vault.sol#262)
        - linkedToken.safeTransfer(msg.sender,linkedTokenAmount) (src/Vault.sol#217)
        - investmentToken.safeTransfer(feeReceiver,fees) (src/Vault.sol#203)
        - linkedToken.safeTransfer(feeReceiver,fees) (src/Vault.sol#203)
        External calls sending eth:
        - execute(priceUpdateData) (src/Vault.sol#185)
                - price = IPythAggregator(_aggregator).getLatestEmaPrice{value: msg.value}(_pythPriceFeed,_priceUpdateData).price (src/AggregatorHelper.sol#33)
        State variables written after the call(s):
        - lpWithdrawn = true (src/Vault.sol#220)
        Vault.lpWithdrawn (src/Vault.sol#47) can be used in cross function reentrancies:
        - Vault.lpWithdraw(bytes[]) (src/Vault.sol#182-222)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities
```

```
Reentrancy in VaultKeeper.performUpkeep(bytes,bytes[]) (src/VaultKeeper.sol#57-69):
        External calls:
        - Vault(vaults[index]).execute(priceUpdateData) (src/VaultKeeper.sol#64)
        State variables written after the call(s):
        - vaults[index] = vaults[vaults.length - 1] (src/VaultKeeper.sol#67)
        VaultKeeper.vaults (src/VaultKeeper.sol#15) can be used in cross function reentrancies:
        - VaultKeeper.addVault(address) (src/VaultKeeper.sol#19-22)
        - VaultKeeper.checkUpkeep(bytes) (src/VaultKeeper.sol#43-54)
        - VaultKeeper.getVaultCount() (src/VaultKeeper.sol#34-36)
        - VaultKeeper.getVaults() (src/VaultKeeper.sol#38-40)
        - VaultKeeper.performUpkeep(bytes,bytes[]) (src/VaultKeeper.sol#57-69)
        - VaultKeeper.removeVault(address) (src/VaultKeeper.sol#24-32)
        - VaultKeeper.vaults (src/VaultKeeper.sol#15)
        - vaults.pop() (src/VaultKeeper.sol#68)
        VaultKeeper.vaults (src/VaultKeeper.sol#15) can be used in cross function reentrancies:
        - VaultKeeper.addVault(address) (src/VaultKeeper.sol#19-22)
        - VaultKeeper.checkUpkeep(bytes) (src/VaultKeeper.sol#43-54)
        - VaultKeeper.getVaultCount() (src/VaultKeeper.sol#34-36)
        - VaultKeeper.getVaults() (src/VaultKeeper.sol#38-40)
        - VaultKeeper.performUpkeep(bytes,bytes[]) (src/VaultKeeper.sol#57-69)
        - VaultKeeper.removeVault(address) (src/VaultKeeper.sol#24-32)
        - VaultKeeper.vaults (src/VaultKeeper.sol#15)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
Vault.lpCancel().linkedTokenAmount (src/Vault.sol#170) is a local variable never initialized
Vault.lpWithdraw(bytes[]).linkedTokenAmount (src/Vault.sol#192) is a local variable never initialized
Vault.lpCancel().investmentTokenAmount (src/Vault.sol#171) is a local variable never initialized
Vault.lpWithdraw(bytes[]).investmentTokenAmount (src/Vault.sol#193) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
ERC1967Utils.upgradeToAndCall(address,bytes) (lib/openzeppelin-contracts-upgradeable/lib/openzeppelin-contracts/contracts/proxy/ERC1967/ERC1967Utils.sol#83-92) ignores return val
ue by Address.functionDelegateCall(newImplementation,data) (lib/openzeppelin-contracts-upgradeable/lib/openzeppelin-contracts/contracts/proxy/ERC1967/ERC1967Utils.sol#88)
ERC1967Utils.upgradeBeaconToAndCall(address,bytes) (lib/openzeppelin-contracts-upgradeable/lib/openzeppelin-contracts/contracts/proxy/ERC1967/ERC1967Utils.sol#173-182) ignores re
turn value by Address.functionDelegateCall(IBeacon(newBeacon).implementation(),data) (lib/openzeppelin-contracts-upgradeable/lib/openzeppelin-contracts/contracts/proxy/ERC1967/ER
C1967Utils.sol#178)
AggregatorHelper.getPrice(address,bytes32,bytes[]) (src/AggregatorHelper.sol#25-35) ignores return value by (None,price,None,None,None) = IChainlinkOracle(_aggregator).latestRoun
dData() (src/AggregatorHelper.sol#30)
VaultKeeper.performUpkeep(bytes,bytes[]) (src/VaultKeeper.sol#57-69) ignores return value by Vault(vaults[index]).execute(priceUpdateData) (src/VaultKeeper.sol#64)
Faucet.claimToken() (src/utils/Faucet.sol#30-35) ignores return value by token.mint(msg.sender,claimAmount[tokens[i]]) (src/utils/Faucet.sol#33)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
Loop condition i < vaults.length (src/VaultKeeper.sol#46) should use cached array length instead of referencing `length` member of the storage array.
 Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#cache-array-length
INFO:Slither:. analyzed (30 contracts with 58 detectors), 21 result(s) found
```

All issues identified by Slither were proved to be false positives or have been added to the issue list in this report.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.