Open Dollar Security Review

Pashov Audit Group

Conducted by: rvierdiiev, ubermensch, peanuts

April 17th 2024 - April 19th 2024

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 **open-dollar/od-relayer** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Open Dollar

Open Dollar is a stablecoin protocol built on Arbitrum. Liquid staking tokens and other Arbitrum-native assets can be deposited into Vaults in order to borrow the OD stablecoin with low-interest loans, and create leveraged positions.

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 - 48224ade8621a8697bb849d7853bd29d3cb7237b

fixes review commit hash - 453222dff7437edae894df2845600ec97a2baf9d

Scope

The following smart contracts were in scope of the audit:

- `CamelotRelayerChild`
- `CamelotRelayerFactory`
- `ChainlinkRelayerChild`
- `ChainlinkRelayerFactory`
- `DenominatedOracleChild`
- `DenominatedOracleFactory`
- `FactoryChild`
- `CamelotRelayer`
- `ChainlinkRelayer`
- `ChainlinkRelayerWithL2Validity`
- `DataConsumerSequencerCheck`
- `DenominatedOracle`
- `Authorizable`
- `Math`

7. Executive Summary

Over the course of the security review, rvierdiiev, ubermensch, peanuts engaged with Open Dollar to review Open Dollar. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	Open Dollar
Repository	https://github.com/open-dollar/od-relayer
Date	April 17th 2024 - April 19th 2024
Protocol Type	Overcollateralized stablecoin

Findings Count

Severity	**Amount**
Medium	2
Low	10
Total Findings	**12**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Incorrect Absolute Value Calculation in _abs Function Leads To DoS on Tokens with More Than 18 Decimals	Medium	Acknowledged
[M-02]	Misleading Validity Checks in read Function of ChainlinkRelayer	Medium	Resolved
[L-01]	An Authorized user can remove himself from authorization	Low	Acknowledged
[L-02]	The onlyFactory modifier is not used in FactoryChild	Low	Acknowledged
[L-03]	CamelotRelayer.read() will revert for newly created pools	Low	Acknowledged
[L-04]	Revert in consult Function with High Decimal Tokens	Low	Acknowledged
[L-05]	Unsafe Casting of Negative Prices in _parseResult	Low	Resolved
[L-06]	Precision Loss in DenominatedOracle Due to Order of Operations	Low	Resolved
[L-07]	Potential Integer Overflow in wmul Function Due to Solidity 0.7.6	Low	Resolved
[L-08]	ChainlinkRelayer has different conditions for getResultWithValidity and read functions	Low	Acknowledged
[L-09]	ChainlinkRelayerFactory can't deploy ChainlinkRelayerWithL2Validity	Low	Resolved
[L-10]	ChainlinkRelayerWithL2Validity.getResultWithValidity should not revert when the sequencer is down	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] Incorrect Absolute Value Calculation in `_abs` Function Leads To DoS on Tokens with More Than 18 Decimals

Severity

Impact: High

Likelihood: Low

Description

The function `_abs(int256 x)` is designed to return the absolute value of the input `x`. However, due to an error in how the ternary operation `x >= 0 ? x : -x` is handled, the result is not assigned back to `x`. This results in the function simply casting `x` to `uint256` without changing its sign. This leads to incorrect behavior when `x` is negative, as the cast of a negative `int256` to `uint256` results in very large numbers, for example `uint256(-1)` returns the maximum value of `uint256`.

```
// @notice Return the absolute value of a signed integer as an unsigned
  // integer
  function _abs(int256 x) internal pure returns (uint256) {
    x >= 0 ? x : -x;
    return uint256(x);
  }
```

This issue affects the `_parseResult` function used in both `ChainlinkRelayer` and `CamelotRelayer` contracts. When `MULTIPLIER` is negative (implying a decimal greater than 18), the faulty `_abs(MULTIPLIER)` returns a large number, which can cause an overflow when raised to the power of 10, leading to a division by zero error and causing a denial of service. Although this scenario is less likely in the `ChainlinkRelayer` due to typical decimal values (8 or 18), it poses a greater risk in the `CamelotRelayer`, which dynamically uses the decimal values of quote tokens.

- ChainlinkRelayer.sol

```
function _parseResult(int256 _chainlinkResult) internal view returns
    (uint256 _result) {
    if (MULTIPLIER == 0) {
      return uint256(_chainlinkResult);
    } else if (MULTIPLIER > 0) {
      return uint256(_chainlinkResult) * (10 ** uint256(MULTIPLIER));
    } else {
      return uint256(_chainlinkResult) / (10 ** _abs(MULTIPLIER));
    }
  }
```

- CamelotRelayer.sol

```
function _parseResult(uint256 _quoteResult) internal view returns
    (uint256 _result) {
    if (MULTIPLIER == 0) {
      return _quoteResult;
    } else if (MULTIPLIER > 0) {
      return _quoteResult * (10 ** uint256(MULTIPLIER));
    } else {
      return _quoteResult / (10 ** _abs(MULTIPLIER));
    }
  }
```

Recommendations

1. Correct the `_abs` function to properly compute the absolute value by ensuring the ternary operation's result is assigned back to `x`. The function should be modified as follows:

```
function _abs(int256 x) internal pure returns (uint256) {
-    x >= 0 ? x : -x;
+    x = x >= 0 ? x : -x;
     return uint256(x);
}
```

Open Dollar comments

None of the collateral types intended for initial protocol launch have a non-standard decimal amount, or more than 18 decimals. We will consider these findings when adding new collateral types, and if necessary create a new relayer or token-wrapper which is compatible with a non-standard decimal amount.

[M-02] Misleading Validity Checks in `read` Function of `ChainlinkRelayer`

Severity

Impact: Medium

Likelihood: Medium

Description

The `read` function of the `ChainlinkRelayer` contract is intended to fetch and return the latest price from a Chainlink feed, reverting if the price is considered invalid. However, the current implementation allows for potentially incorrect behavior due to the misuse of logical operators in the validity check. The function checks that the price `_aggregatorResult` is either non-zero or the data is fresh, using an OR (`||`) condition. This allows the function to return a zero price if the data is not stale, and conversely, to return a stale price as long as it is non-zero. Both scenarios are likely unintended and can lead to misleading results being accepted as valid.

The sister function `getResultWithValidity` uses an AND (`&&`) condition for similar checks, suggesting that the intention was to require both conditions (non-zero and freshness) to be true for the data to be considered valid.

Recommendations

Modify the `read` function's validity check to use an AND (`&&`) operator instead of an OR (`||`), aligning it with the logical conditions used in `getResultWithValidity`. This change ensures that only non-zero, fresh prices are considered valid. The corrected part of the function should look like this:

```
function read() public view virtual returns (uint256 _result) {
    // Fetch values from Chainlink
    (
      ,
      int256_aggregatorResult,
      ,
      uint256_aggregatorTimestamp,

    ) = chainlinkFeed.latestRoundData(

    // Revert if price is invalid
-    require(_aggregatorResult != 0 || _isValidFeed
- (_aggregatorTimestamp), 'InvalidPriceFeed');
+    require(_aggregatorResult != 0 && _isValidFeed
+ (_aggregatorTimestamp), 'InvalidPriceFeed');
    // Parse the quote into 18 decimals format
    _result = _parseResult(_aggregatorResult);
  }
```

Open Dollar comments

Implemented the recommended change.

8.2. Low Findings

[L-01] An Authorized user can remove himself from authorization

In Authorizable.sol, there is a function `removeAuthorization()` whereby an Authorized user can remove authorization from an authorized account.

```
function removeAuthorization(address _account) external virtual isAuthorized {
    _removeAuthorization(_account);
  }
```

There is no check to make sure the authorized user cannot remove himself. If the authorized user removes himself accidentally and there are no more users with authorization, there is no way to get back authorization since someone needs to be authorized to call `addAuthorization()`

This will affect the deployment of the Relayers as it is under the `isAuthorized` modifier.

```
function deployChainlinkRelayer(
    address _aggregator,
    uint256 _staleThreshold
> ) external isAuthorized returns (IBaseOracle _chainlinkRelayer) {
    _chainlinkRelayer = IBaseOracle(address(new ChainlinkRelayerChild
      (_aggregator, _staleThreshold)));
    relayerId++;
```

Make sure the authorized user cannot withdraw his authorization and make sure there is always someone with authorization.

Open Dollar comments

We recognize that an authorized account can be removed, and further modifications could be prevented. There is no risk to the protocol since a new factory could be deployed at any time if authorization is lost.

[L-02] The `onlyFactory` modifier is not used in FactoryChild

When creating a child contract from the RelayerFactory, the child contract inherits FactoryChild.

```
contract ChainlinkRelayerChild is ChainlinkRelayer, FactoryChild {
  // --- Init ---

  /**
   * @param  _aggregator The address of the aggregator to relay

       * @param  _staleThreshold The threshold in seconds to consider the aggregator stale
   */
  constructor(address _aggregator, uint256 _staleThreshold) ChainlinkRelayer
    (_aggregator, _staleThreshold) {}
}
```

This FactoryChild is an abstract contract that has an onlyFactory() modifier, but it is not used in the protocol.

```
/// @dev Verifies that the contract is being deployed by a contract address
  constructor() {
    factory = msg.sender;
  }

  // --- Modifiers ---

  ///@dev Verifies that the caller is the factory
  modifier onlyFactory() {
    require(msg.sender == factory, 'CallerNotFactory');
    _;
  }
```

Consider using the modifier like how it is done in AlgebraPoolDeployer when deploying a new AlgebraPool.

```
/// @inheritdoc IAlgebraPoolDeployer
  function deploy(
    address dataStorage,
    address _factory,
    address token0,
    address token1
> ) external override onlyFactory returns (address pool) {
    parameters = Parameters(
      {dataStorage:dataStorage,
      factory:_factory,
      token0:token0,
      token1:token1}
    );
    pool = address(new AlgebraPool{salt: keccak256(abi.encode(token0, token1))}
      ());
  }
}
```

Open Dollar comments

We do not believe the onlyFactory modifier should be implemented here, since deploying new factory children is not intended to be a protected function.

[L-03] CamelotRelayer.read() will revert for newly created pools

In CamelotRelayer.sol, `getResultWithValidity()` will get the price of the token by calling `DataStorageLibrary.consult` and giving a `QUOTE_PERIOD`

```
function getResultWithValidity() external view returns
  (uint256 _result, bool _validity) {
    // TODO: add catch if the pool doesn't have enough history - return false

    // Consult the query with a TWAP period of QUOTE_PERIOD
    int24 _arithmeticMeanTick = DataStorageLibrary.consult
      (algebraPool, QUOTE_PERIOD);
```

This `QUOTE_PERIOD` is the TWAP period that the protocol intends to use to prevent price manipulation attacks, eg 15 mins (900).

In the event that the pool is newly created, `QUOTE_PERIOD` will not work, and `getResultWithValidity()` will revert. It seems that the intention of the function, noted from the

TODO comment, is to return false instead of reverting.

Consider wrapping the function in a try/catch to return a `false` validity for newly created pools

Open Dollar comments

We will ensure that a new oracle returns a valid price feed before it is added to the protocol or used in production in any way. Newly created pools, which do not have a valid price feed, are not intended to be used by the protocol.

[L-04] Revert in `consult` Function with High Decimal Tokens

The `consult` function in the `DataStorageLibrary` interacts with the `AlgebraPool` to retrieve time-weighted average ticks based on specified periods. It constructs an array of `secondAgos` to fetch historical tick cumulatives. However, this function can revert with an error (EvmError: InvalidFEOpcode which is mostly returned for mathematical errors) when interacting with tokens that have more than 18 decimals. The revert occurs because the `getTimepoints` function of the algebra pool might not handle the precision required for tokens with unusually high decimal counts, especially when calculating differences in tick values over specific periods.

Ensure validating the decimal count of tokens is 18 or less before attempting operations that involve the algebra pool.

Open Dollar comments

None of the collateral types intended for initial protocol launch have a non-standard decimal amount, or more than 18 decimals. We will consider these findings when adding new collateral types, and if necessary create a new relayer or token-wrapper which is compatible with a non-standard decimal amount.

[L-05] Unsafe Casting of Negative Prices in `_parseResult`

The `_parseResult` function performs a conversion from `int256` to `uint256` on Chainlink feed results. Chainlink price feeds may return negative prices under certain conditions. Currently, the function does not check the sign of the price before casting it to `uint256`. This can lead to incorrect results, as casting a negative `int256` to `uint256` results in a very large number close to the maximum value of `uint256`, potentially leading to unexpected behavior or financial inaccuracies in the system.

To prevent incorrect data handling and potential vulnerabilities associated with overflow, it is recommended to modify the `_parseResult` function to include a check for negative values before casting. If a negative price is detected, the function should revert or handle the case appropriately. Here is a suggested implementation:

```
function _parseResult(int256 _chainlinkResult) internal view returns
  (uint256 _result) {
+    require(_chainlinkResult >= 0, "Negative price value not allowed");

    if (MULTIPLIER == 0) {
        return uint256(_chainlinkResult);
    } else if (MULTIPLIER > 0) {
        return uint256(_chainlinkResult) * (10 ** uint256(MULTIPLIER));
    } else {
        return uint256(_chainlinkResult) / (10 ** _abs(MULTIPLIER));
    }
}
```

Open Dollar comments

Implemented the recommended change.

[L-06] Precision Loss in `DenominatedOracle` Due to Order of Operations

The `getResultWithValidity` and `read` functions in the `DenominatedOracle`, when handling inverted price sources, perform a division before a multiplication. This leads to a loss of precision in the resulting value. Specifically, the issue arises in scenarios where the inversion is necessary, such as calculating the value of ETH/USDT when the price feed directly provides USD/ETHT. By dividing before multiplying, the calculation loses precision due to integer division truncating any fractional part before it can be offset by the subsequent multiplication.

For example, for an ETH/USDT price of approximately 3000*1e18, the ideal calculation for the inverted price should preserve as much precision as possible: 10^18 x 10^18/ (3000 x 10^18) = `333 333 333 333 333.333333...` However, the current implementation first performs the division, which truncates the result and leads to a loss of decimal precision. Therefore, it will use this `333 333 333 333 333` instead.

Modify the `getResultWithValidity` function to multiply by `_denominationPriceSourceValue` before dividing by `_priceSourceValue` when `INVERTED` is true. This adjustment ensures that the multiplication, which can potentially increase the result magnitude, is done prior to the division, thus preserving more precision. Here is the corrected portion of the function:

```
function getResultWithValidity() external view returns
  (uint256 _result, bool _validity) {
    (
      uint256 _priceSourceValue,
      bool _priceSourceValidity
    ) = priceSource.getResultWithValidity(
    (
      uint256 _denominationPriceSourceValue,
      bool _denominationPriceSourceValidity
    ) =
      denominationPriceSource.getResultWithValidity();

    if (INVERTED) {
      if (_priceSourceValue == 0) return (0, false);
      _result = WAD.wmul(_denominationPriceSourceValue).wdiv(_priceSourceValue);
    } else {
      _result = _priceSourceValue.wmul(_denominationPriceSourceValue);
    }

    _validity = _priceSourceValidity && _denominationPriceSourceValidity;
}
```

Same for the `read` function.

Open Dollar comments

Implemented the recommended change.

[L-07] Potential Integer Overflow in `wmul` Function Due to Solidity 0.7.6

The `wmul` function used in the contract for multiplying two values and then dividing by a predefined constant (`WAD`) is susceptible to integer overflow. This is primarily because the contract is compiled with Solidity version 0.7.6, which does not automatically check for overflows and underflows as versions 0.8 and above do. In this function, the multiplication `_x * _y` could exceed the maximum value for a `uint256` if both `_x` and `_y` are large enough, leading to uncaught overflows.

Consider upgrading to Solidity version 0.8.0 or newer. These versions include automatic overflow and underflow checking, which would inherently prevent this issue without additional gas costs for manual checks. It is also possible to use SafeMath library to prevent the issue.

Open Dollar comments

We have implemented a solution which prevents overflow using a recommended approach from the Open Zeppelin library SafeMath. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v3.3/contracts/math/SafeMath.sol

All wmul function calls use the Math library, which now checks for overflows.

[L-08] ChainlinkRelayer has different conditions for getResultWithValidity and read functions

`ChainlinkRelayer.getResultWithValidity` function allows only positive price: `_validity = _aggregatorResult > 0 && _isValidFeed(_aggregatorTimestamp);`.

`ChainlinkRelayer.read` function allows negative prices: `require(_aggregatorResult != 0 || _isValidFeed(_aggregatorTimestamp), 'InvalidPriceFeed');`.

Both those functions are the same, so the check should be the same, and negative prices should not be allowed in both cases.

Open Dollar comments

No changes are needed. The negative value is checked during _parseResult, therefore the price returned cannot be negative.

[L-09] ChainlinkRelayerFactory can't deploy ChainlinkRelayerWithL2Validity

ChainlinkRelayerWithL2Validity extends ChainlinkRelayer and provides L2 sequencer check. Currently, it can't be deployed with ChainlinkRelayerFactory and there is no separate factory for it, as ChainlinkRelayerFactory allows to deploy ChainlinkRelayerChild only, which is ChainlinkRelayer.

```
function deployChainlinkRelayer(
    address _aggregator,
    uint256 _staleThreshold
  ) external isAuthorized returns (IBaseOracle _chainlinkRelayer) {
    _chainlinkRelayer = IBaseOracle(address(new ChainlinkRelayerChild
      (_aggregator, _staleThreshold)));
    relayerId++;
    relayerById[relayerId] = address(_chainlinkRelayer);
    emit NewChainlinkRelayer(address
      (_chainlinkRelayer), _aggregator, _staleThreshold);
  }
```

Open Dollar comments

Implemented the recommended change.

[L-10] ChainlinkRelayerWithL2Validity.getResultWithValidity should not revert when the sequencer is down

ChainlinkRelayerWithL2Validity.getResultWithValidity function is designed to not revert, when the result is not valid, it should just return `false` as validity. But in case the sequencer is down, then the function reverts, which may revert integrations.

```
function getResultWithValidity() public view override returns
  (uint256 _result, bool _validity) {
    require(getSequencerFeedValidation(), 'SequencerDown');
    (_result, _validity) = super.getResultWithValidity();
  }
```

Open Dollar comments

Implemented the recommended change.