Reserve Protocol
Security Review

Cantina Managed review by:
Jonah1005, Lead Security Researcher
Xmxanuel, Security Researcher

March 18, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

The Reserve protocol is the first platform that allows for the permissionless creation of asset-backed, yield-bearing and overcollateralized stablecoins on Ethereum. The end goal of the Reserve protocol is to provide highly scalable, decentralized, stable money in contrast to volatile cryptocurrencies such as Bitcoin and Ether.

From Mar 4th to Mar 5th the Cantina team conducted a review of moon-ramp on commit hash 49d088fe. The team identified a total of **17** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 4
- Low Risk: 4
- Gas Optimizations: 0
- Informational: 8

3 Findings

3.1 High Risk

3.1.1 `amountOut` lacks decimal conversion, leading to mispricing for non-18 decimal tokens

Severity: High Risk

Context: MoonRamp.sol#L156-L191

Description: In the `getAmountIn` function, all internal calculations are done in 18 decimals. However, `amountOut` isn't normalized in the function, resulting in an incorrect price when tokens are not in 18 decimals.

Recommendation: Consider normalizing `amountOut` with `UNIT_RWRD`.

Reserve: Fixed in commit 8b64619f.

3.2 Medium Risk

3.2.1 Mitigate reentrancy attacks on `incentiveDistributor` with `nonReentrant` in `MoonRamp`

Severity: Medium Risk

Context: MoonRamp.sol#L218-L229

Description: In the bid function, the `MoonRamp` function triggers `incentiveDistributor.onBid` after the `_safeMint` of ERC721. The receiver can interact with the `MoonRamp` in the callback function before the `incentiveDistributor` processes the incentives. This could lead to the `IncentiveDistributor` fetching the wrong value and subsequently having an incorrect incentive distribution.

Recommendation: Consider adding the `nonReentrant` modifier to functions that modify states, such as `bid`, `bail` and `lunch`.

Reserve: Fixed in commit 92e2d188.

3.2.2 `incentiveDistributor` can be bypassed by forcing the external call in a try-catch clause to revert

Severity: Medium Risk

Context: MoonRamp.sol#L223-L226, MoonRamp.sol#L223-L226, MoonRamp.sol#L278-L282

Description: The `MoonRamp` contract utilizes a try-catch clause when calling `incentiveDistributor`. In the event that the `incentiveDistributor` is malfunctioning and consistently reverts transactions, the `MoonRamp` itself would not be susceptible to a Denial-of-Service (DoS) attack. However, malicious users could deliberately trigger a revert by carefully adjusting the gas limit of the transaction.

Let's assume the `incentiveDistributor` consumes 80000 gas in the `onBail` function. Malicious users could induce a revert in `incentiveDistributor.onBail` by leaving less than 80000 gas at the end of the bail function.

Assume a `incentiveDistributor` is a basic reward distribution contract that fetches `rampInfo` from the `moonRamp` contract and stores it locally.

```solidity
contract BasicIncentiveDistributor {
    MoonRamp mr;
    struct RampStruct {
        address owner;
        uint256 collateral;
        uint256 reward;
        uint256 bailTime;
        uint256 updateTime;
        uint256 lastSpotPrice;
        uint256 pendingRewards;
    }
    mapping(uint256 => RampStruct) public ramps;
    constructor(address moonRamp) {
        mr = MoonRamp(moonRamp);

    }
    function ownerOf(uint256 id) external view returns (address) {
        return ramps[id].owner;
    }

    function onBid(uint256 id) external {
        uint gasBefore = gasleft();
        require(msg.sender == address(mr), "IncentiveDistributor: not authorized");
        (address owner, MoonRamp.Ramp memory _ramp) = mr.rampInfo(id);
        ramps[id] = RampStruct({
            owner: owner,
            collateral: _ramp.collateral,
            reward: _ramp.reward,
            bailTime: _ramp.bailTime,
            updateTime: block.timestamp,
            lastSpotPrice: mr.lastSpotPrice(),
            pendingRewards: mr.pendingRewardsOut()
        });
    }
    function onBail(uint256 id) external {

    }
    function onLaunch(uint256 id) external {

    }
}
```

The malicious user can provide slightly less gas and prevent the `rewardDistributor` storing critical information:

```
function testMoonRampBasicIncentiveDistributor() public {
    MoonRamp mr = defaultMoonRamp();
    pp.setDecayConstant(ONE_PCT_PER_WEEK);
    pp.setBeta(TEN_BPS_PER_TOKEN);
    lfp.setLaunchFee(1.0005e18);

    // approve collateral token transfer
    defaultCollateralToken.approve(address(mr), type(uint256).max);
    // endow MoonRamp with reward tokens
    defaultRewardToken.transfer(address(mr), 1_000e18);

    vm.warp(444);
    assertTrue(mr.lastBidTime() != block.timestamp);

    uint256 pendingRewardsOutBefore = mr.pendingRewardsOut();
    uint256 collBalThisBefore = defaultCollateralToken.balanceOf(address(this));
    (uint256 amountIn, uint256 fee, uint256 nextSpotPrice) = mr.getAmountIn(137e18, block.timestamp);
    BasicIncentiveDistributor distributor = new BasicIncentiveDistributor(address(mr));
    mr.setIncentiveDistributor(address(distributor));
    uint previousGas = gasleft();
    uint256 id = mr.bid(137e18, 20_000e18);
    assertEq(distributor.ownerOf(id), address(this));
}

function testMoonRampBasicIncentiveDistributorRevert() public {
    MoonRamp mr = defaultMoonRamp();
    pp.setDecayConstant(ONE_PCT_PER_WEEK);
    pp.setBeta(TEN_BPS_PER_TOKEN);
    lfp.setLaunchFee(1.0005e18);

    // approve collateral token transfer
    defaultCollateralToken.approve(address(mr), type(uint256).max);

    // endow MoonRamp with reward tokens
    defaultRewardToken.transfer(address(mr), 1_000e18);

    vm.warp(444);
    assertTrue(mr.lastBidTime() != block.timestamp);

    uint256 pendingRewardsOutBefore = mr.pendingRewardsOut();
    uint256 collBalThisBefore = defaultCollateralToken.balanceOf(address(this));
    (uint256 amountIn, uint256 fee, uint256 nextSpotPrice) = mr.getAmountIn(137e18, block.timestamp);
    BasicIncentiveDistributor distributor = new BasicIncentiveDistributor(address(mr));
    mr.setIncentiveDistributor(address(distributor));
    // uint previousGas = gasleft();
    uint256 id = mr.bid{gas: 387414 - 5000}(137e18, 20_000e18);
    // log: gas used: 387414
    // console2.log("gas used:", previousGas - gasleft());
    assertEq(distributor.ownerOf(id), address(0));
}
```

Recommendation: The try-catch is utilized to mitigate the potential Denial-of-Service (DoS) scenario in case the distributor behaves maliciously. However, considering that the distributor can be replaced by the controller and the incentive distribution can be critical in most use cases, I recommend removing the try-catch clause.

Reserve: Fixed in commit bc58a689.

3.2.3 `ChainlinkParamProvider`: Chainlink's `latestRoundData` may return stale prices or incorrect results

Severity: Medium Risk

Context: ChainlinkParamProvider.sol#L41

Description: The Chainlink oracle can return stale prices or incorrect results if it stops working correctly.

Recommendation: We recommend using common `Chainlink` oracle checks for `latestRoundData` and implementing measures to handle a stale floor price. The `answeredInRound` return value is deprecated in the latest version but some Chainlink Oracles might still use it.

```
(uint80 roundID ,int256 price,, uint256 timestamp, uint80 answeredInRound) = feed.latestRoundData();
require(price > 0, "Chainlink price <= 0");
require(answeredInRound >= roundID, "Stale price");
require(timestamp != 0, "Round not complete");
```

In the current design, the `price` is required to calculate the `floorPrice` which is needed to calculate the `currentPrice` in `MoonRamp`. A revert here would result in blocking new `MoonRamp.bid` transaction.

We recommend discussing the issue within your team, if the bid transaction should revert in this scenario or if it is acceptable to continue working with a slightly outdated `floorPrice`.

Reserve: After talking this over in detail within the Reserve team, the consensus is that even a stale price is likely better than returning zero or reverting, since the idea is that it's just being used as a lower bound with something like a 50% multiplier.

The consensus is also to prefer returning zero if the price goes negative. So, at the end of the day, no change to the logic but I'll enhance the comments a bit (see commit e6d9a8ad).

3.2.4 Anyone can increase the `rewardToken` supply used for ramps by transferring tokens to the `MoonRamp` contract

Severity: Medium Risk

Context: MoonRamp.sol#L204

Description: The `MoonRamp` contract allows users to `bid` on reward tokens based on a continuously changing price influenced by time and previous `bids`.

New bids can happen as long as the `MoonRamp` contract has enough reward tokens. This is verified by checking the `ERC20` balance:

```
if (newPendingRewardsOut > rewardToken.balanceOf(address(this)))  revert InsufficientRewardBalance();
```

This means anyone can increase the initial supply by sending more reward tokens to the contract. The `MoonRamp` contract includes an `incentiveDistributor` for all new bids. There might be a financial incentive to send more reward tokens to the contract and immediately bid afterwards, long after the initial token distribution has concluded.

Recommendation: Consider adding a permissioned `topUp` function, only callable by the `controller`, to increase the `rewardTokenSupply`. An additional storage variable should track the `rewardTokenSupply` usage to check in the `bid` functions if enough `rewardTokens` are available.

Reserve: Added an `addRewardTokens` function in commit 4ec2d319.

3.3 Low Risk

3.3.1 Precision loss in `betaQ` results in a lower bid price

Severity: Low Risk

Context: MoonRamp.sol#L173

Description: The `bid` function would not increase the price of the reward token if `amountOut` is small. The `betaQ` is calculated as:

```
uint256 betaQ = Math.mulDiv(params.beta, amountOut, UNIT_RWRD, Math.Rounding.Ceil);
```

The `betaQ` would be `1` when `amountOut` is small leading to `increaseFactor == 1`.

The following proof of concept script shows that bidding would not change the price when `amountOut == 1e3`. Consequently, splitting a real bid into multiple smaller bids would yield users a better price.

```
function testSplitAmountBid_Small() public {
    MoonRampConfig memory conf = defaultConfig();
    conf.initialPrice = 1e18;
    MoonRamp mr = defaultMoonRampWithConfig(conf);
    pp.setDecayConstant(ONE_PCT_PER_WEEK);
    pp.setBeta(TEN_BPS_PER_TOKEN);
    lfp.setLaunchFee(1.0005e18);
    // approve collateral token transfer
    defaultCollateralToken.approve(address(mr), type(uint256).max);

    // endow MoonRamp with reward tokens
    defaultRewardToken.transfer(address(mr), 1_000_000 ether);

    (uint256 amountIn, uint256 fee, uint256 nextSpotPrice) = mr.getAmountIn(1e18, block.timestamp);
    (uint splitAmountIn, uint splitFee, uint splitNextSpotPrice) = mr.getAmountIn(1e3, block.timestamp);
     // spot price does not change when amountOut is small.
    assertEq(splitNextSpotPrice, conf.initialPrice);
    assertLt(splitNextSpotPrice, nextSpotPrice);
    assertLt(splitAmountIn * (1e18 / 1e3), amountIn);
}
```

Recommendation: Conducting such "attacks" would not result in any profits considering the cost of executing transactions. However, we shall be aware of the potential risks. The transaction costs on L2 chains are expected to be low in the future.

Reserve: Acknowledged. Leaning towards not making any change at the moment besides documentation for this one.

3.3.2 Changing the `incentiveDistributor` can result in a `ramp` using two different `incentiveDistributors`

Severity: Low Risk

Context: MoonRamp.sol#L290

Description: The `incentiveDistributor` is an optional contract which is called on the main operations:

```
interface IncentiveDistributor {
    function onBid(uint256 id) external;
    function onBail(uint256 id) external;
    function onLaunch(uint256 id) external;
}
```

The `MoonRamp` contract has a function to change the `setIncentiveDistributor`.

However, this can result in the following: for some users `bid` has been called on the old `incentiveDistributor` and `launch/bail` will be called on the new one.

Recommendation: The `incentiveDistributor` has to consider this fact in a migration. A missing `onBid` call should not lead to disadvantages in the `onLaunch` or `onBail` call.

Reserve: To simplify incentive distributor design considerations, the ability to change the incentive distributor in the `MoonRamp` has been removed in commit 4ca91d39. If upgradability is desired, the auction controller should use an upgradability pattern for the incentive distributor itself (for example a delegate-call proxy).

3.3.3 Prevent `MoonRamp.bid` and `MoonRamp.bail` transactions in the same `block`

Severity: Low Risk

Context: MoonRamp.sol#L241

Description: In `MoonRamp` users can `bid` on the `currentPrice` by locking `collateral` in the contract. The amount of `rewardTokens` users are willing to `bid` will impact the next price by a price increase based on the amount of `rewardTokens` they inted to buy.

After the `bid` users can either call `launch` which will exchange the `collateralToken` for the `rewardTokens` minus a fee, or choose to call `bail` after a `bailDelay`. In a `bail` scenario the `collateralToken` are returned to the user with no extra cost.

The `bailDelay` is used to prevent attacks aimed at artificially increasing the price by calling `bid` and immediately afterward `bail`. There would be no extra cost involved and without a `bailDelay` attackers could use flashloans or perform a `bid/bail` in a loop inside a transaction.

However, it is still possible to set the `bailDelay` to zero.

Recommendation: Change the `BailDelay` check in the `bail` function to:

```
- if (block.timestamp < ramp.bailTime) revert BailDelayNotOver();
+ if (block.timestamp <= ramp.bailTime) revert BailDelayNotOver();
```

The `bailTime` is set in the bid function with `bailTime = block.timestamp + bailDelay`. Alternatively, if the `bailDelay` stays a immutable system parameter defined in the constructor. The constructor could enforce that `bailDelay > 0`.

Reserve: `bailDelay==0` should be fine if `beta==0`, which is a valid potential use-case (this reduces to just selling tokens in a Dutch auction).

3.3.4 `MoonRamp.currentPrice` view function should revert if `params.decayConstant > ONE`

Severity: Low Risk

Context: MoonRamp.sol#L151

Description: The `price` of the `rewardToken` decreases over time in `MoonRamp` based on a `decayConstant` which provided by the `parameterProvider` contract.

The `decayConstant` has to be smaller than `1e18` (`ONE`) to decrease the price. The `bid` function has the following check:

```
if (params.decayConstant > ONE) revert DecayConstantTooLarge();
```

However, the existing `currentPrice` view function does not consider this and would return a price calculated with `decayConstant > ONE`.

Recommendation: The `currentPrice` view function should revert if the `decayConstant > ONE` because the price would be incorrect and a `bid` transaction would revert as well.

Reserve: Fixed in commit 91262d5d.

3.4 Informational

3.4.1 Add sanity checks for `reward` and `collateral` token in constructor

Severity: Informational

Context: MoonRamp.sol#L124

Description: Currently, no additional checks in the `MoonRamp` constructor are applied for the passed ERC20 addresses `collateral` and `rewardToken`.

Recommendation: Add an upper limit for supported `decimals` to avoid `weird` ERC20 behavior.

3.4.2 Analyze the system behaviour for a potential price decrease in `bail`

Severity: Informational

Context: MoonRamp.sol#L237

Description: In the current MoonRamp design a `bail` has no implications on the price. The user simply receives their collateral back.

The price is only decreases over time and increases in the event of a `bid` action. The increase is based on the amount of reward tokens, the current price and the variable `beta` as an increase factor.

Recommendation: Analyze the implications if a `bail` would decrease the price, essentially serving as the inverse operation to a bid increase. The lower price might attract new bidder especially since the previous user decide to not execute their `rewardTokens` purchase.

3.4.3 Analyze if adding a 5th term to the power series in `MoonRamp.calcFactors` can increase the precision

Severity: Informational

Context: MoonRamp.sol#L345

Description: The price increase in `MoonRamp` after a `bid` is defined by a price change function: `P(q)` where `q` represents the amount of `rewardToken` (for more details see MoonRamp README).

To calculate the required amount of `collateralToken` for `q` `rewardTokens`, it is necessary to integrate over the price function. Instead of implementing the integral directly, the contract itself expresses it with a `slippageFactor` which is multipled with the `lastestPrice` and the amount of `rewardsToken`.

The `MoonRamp.calcFactors` function calculates the slippage factor. However, for smaller factors of `betaQ` of `6e14`. A power series approximation is used.

```
unchecked {
    slippageFactor =   ONE
        + betaQ / 2
        + (betaQ * betaQ) / (6 * ONE)
        + (betaQ * betaQ * betaQ) / (24 * ONE**2);
}
```

Recommendation: Currently 4 terms of the power series are used. Analyze whether a 5th term would increase the precision.

3.4.4 Only use `uint48` or `uint256` to store `block.timestamp` for consistency

Severity: Informational

Context: MoonRamp.sol#L102)

Description: Currently, `uint48` and `uint256` are used for `block.timestamp`.

- MoonRamp.sol#L102:

```
uint48  public lastBidTime;
```

- MoonRamp.sol#L151:

```
 function currentPrice(uint256 timestamp) public view returns (uint256)
```

  Together with the `bailDelay` for a time interval.

- MoonRamp.sol#L59:

```
uint256 public immutable bailDelay;
```

Recommendation: Use only `uint256` or `uint48` for `block.timestamp`. Normally, `uint48` is sufficient.

3.4.5 Increase test coverage for edge cases in the `MoonRamp` contract

Severity: Informational

Context: MoonRamp.sol#L34

Description: Currently the `tests` are missing edge case and different token precision tests.

Recommendation: Consider adding tests for the following:

- `beta` parameter of 0.
- `launchFee` less than `1e18`.
- Different collateral token precision than `1e18`.
- Different reward token precision than `1e18`.

Reserve:

- `beta == 0` and `launchFee < 0` tested in commit 308faff9(ignore the commit message for it contains a typo).
- Basic tests for tokens with different decimals were added in the commit addressing the decimals bug in `getAmountOut` in commit 8b64619f.

3.4.6 Consider "_" prefix for internal functions

Severity: Informational

Context: MoonRamp.sol#L323

Description: Currently, in the `MoonRamp` contract internal functions don't have an "_" prefix. However, the contract inherits from OZ library contracts which use this pattern.

Recommendation: Add _ prefix to internal functions for consistency reasons.

Reserve: Fixed in commit 2aec5f2e

3.4.7 Lack of events in `SettableParamProvider`

Severity: Informational

Context: SettableParamProvider.sol#L7

Description: The `SettableParamProvider` contract lacks the use of events for the main operations.

Recommendation: Consider adding events for the main operations.

3.4.8 Use the same variable name for the `rampId` for consistency

Severity: Informational

Context: MoonRamp.sol#L202

Description: There a two different names used for the `rampId` in the `MoonRamp` contract:

- `rampId`: MoonRamp.sol#L264
- `id`: MoonRamp.sol#L202

Recommendation: Use the same variable name for the ramp id.

Reserve: Fixed in commit 86f1e9ed.