Reserve Protocol - Invitational Findings & Analysis Report

2023-11-13

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Reserve Protocol smart contract system written in Solidity. The audit took place between July 25 — August 4 2023.

Following the C4 audit, 3 wardens (ronnyx2017, bin2chen and RaymondFam) reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit report.

Wardens

In Code4rena's Invitational audits, the competition is limited to a small group of wardens; for this audit, 7 wardens contributed reports::

1. ronnyx2017
2. bin2chen
3. RaymondFam
4. 0xA5DF
5. auditor0517
6. sces60107
7. carlitox477

This audit was judged by cccz.

Final report assembled by liveactionllama and thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 18 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 15 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 6 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 2 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the [C4 Reserve Protocol Audit repository](), and is composed of 51 smart contracts written in the Solidity programming language and includes approximately 3000 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](), specifically our section on [Severity Categorization]().

High Risk Findings (3)

[H-01] CBEthCollateral and AnkrStakedEthCollateral _underlyingRefPerTok is incorrect

Submitted by ronnyx2017, also found by 0xA5DF

Lines of Code

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/cbeth/CBETHCollateral.sol#L67-L69
https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/ankr/AnkrStakedEthCollateral.sol#L58-L61

The `CBEthCollateral._underlyingRefPerTok()` function just uses `CBEth.exchangeRate()` to get the ref/tok rate. The `CBEth.exchangeRate()` can only get the conversion rate from cbETH to staked ETH2 on the coinbase. However as the docs `https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/2023-07-reserve/protocol/contracts/plugins/assets/cbeth/README.md` the ref unit should be ETH. The staked ETH2 must take a few days to unstake, which leads to a premium between ETH and cbETH.

And the `AnkrStakedEthCollateral` and `RethCollateral` has the same problem. According to the ankr docs, unstake eth by Flash unstake have to pay a fee, 0.5% of the unstaked amount. https://www.ankr.com/docs/liquid-staking/eth/unstake/

Impact

The `_underlyingRefPerTok` will return a higher ref/tok rate than the truth. And the premium is positively correlated with the unstake delay of eth2. When the unstake queue suddenly increases, the attacker can uses cbeth to issue more rtokens. Even if the cbETH has defaulted, the CBEthCollateral will never mark the state as DISABLED because the `CBEth.exchangeRate()` is updated by coinbase manager and it only represents the cbETH / staked eth2 rate instead of the cbETH/ETH rate.

Proof of Concept

For example, Now it's about 17819370 block high on the mainnet, and the `CBEth.exchangeRate()` (https://etherscan.io/token/0xbe9895146f7af43049ca1c1ae358b0541ea49704#readProxyContract#F12) is 1.045264058480813188, but the chainlink price feed for cbETH/ETH(https://data.chain.link/ethereum/mainnet/crypto-eth/cbeth-eth) is 1.0438.

Recommended Mitigation Steps

Use the `cbETH/ETH` oracle to get the `cbETH/ETH` rate.

Or, the ref unit for the collateral should be the staked eth2.

Assessed type

Context

tbrent (Reserve) commented:

> This feels like a duplicate of **#32**. The root cause is an incorrect reference unit. The reference unit should be staked eth2, as indicated here.

pmckelvy1 (Reserve) confirmed

ronnyx2017 (warden) commented:

> This issue and **32** explain the misuse of tar unit and ref unit in staked eth related assets from different perspectives. The root cause is same, that 1 staked eth2 != 1 eth. This issue assumes that the ref token and target is all eth, which is referred to in the docs. So the error should be in the function `_underlyingRefPerTok`. But issue 32 assumes that the ref unit should be staked eth2 and the target unit is eth. So it needs to modify function `targetPerRef`. I also have mentioned this mitigation in the `Recommended Mitigation Steps` section of the current issue:
>
> ```
> Or, the ref unit for the collateral should be the staked eth2.
> ```

cccz (judge) increased severity to High

Reserve Mitigated:

> Fixes units and price calculations in cbETH, rETH, ankrETH collateral plugins.
> PR: https://github.com/reserve-protocol/protocol/pull/899

Status: Mitigation confirmed. Full details in reports from ronnyx2017, bin2chen and RaymondFam.

[H-02] CurveVolatileCollateral Collateral status can be manipulated by flashloan attack

Submitted by ronnyx2017

Attacker can make the CurveVolatileCollateral enter the status of IFFY/DISABLED. It will cause the basket to rebalance and sell off all the CurveVolatileCollateral.

Proof of Concept

The `CurveVolatileCollateral` overrides the `_anyDepeggedInPool` function to check if the distribution of capital is balanced. If the any part of underlying token exceeds the expected more than `_defaultThreshold`, return true, which means the volatile pool has been depeg:

```
    uint192 expected = FIX_ONE.divu(nTokens); // {1}
    for (uint8 i = 0; i < nTokens; i++) {
        uint192 observed = divuu(vals[i], valSum); // {1}
        if (observed > expected) {
            if (observed - expected > _defaultThreshold) return true
        }
    }
```

And the coll status will be updated in the super class `CurveStableCollateral.refresh()`:

```
    if (low == 0 || _anyDepeggedInPool() || _anyDepeggedOutsidePool(
        markStatus(CollateralStatus.IFFY);
    }
```

The attack process is as follows:

1. Assumption: There is a CurveVolatileCollateral bases on a TriCrypto ETH/WBTC/USDT, and the value of them should be 1:1:1, and the

`_defaultThreshold` of the CurveVolatileCollateral is 5%. And at first, there are 1000 USDT in the pool and the pool is balanced.

2. The attacker uses flash loan to deposit 500 USDT to the pool. Now, the USDT distribution is `1500/(1500+1000+1000) = 42.86%`.
3. Attacker refresh the CurveVolatileCollateral. Because the USDT distribution - expected = 42.86% - 33.33% = 9.53% > 5% _defaultThreshold. So CurveVolatileCollateral will be marked as IFFY.
4. The attacker withdraw from the pool and repay the USDT.
5. Just wait `delayUntilDefault`, the collateral will be marked as defaulted by the `alreadyDefaulted` function.

```
    function alreadyDefaulted() internal view returns (bool) {
        return _whenDefault <= block.timestamp;
    }
```

Recommended Mitigation Steps

I think the de-pegged status in the volatile pool may be unimportant. It will be temporary and have little impact on the price of outside lp tokens. After all, override the `_anyDepeggedOutsidePool` to check the lp price might be a good idea.

Assessed type

Context

tbrent (Reserve) confirmed

Reserve Mitigated:

> Removes `CurveVolatileCollateral`.
> PR: https://github.com/reserve-protocol/protocol/pull/896

Status: Mitigation confirmed. Full details in reports from ronnyx2017, RaymondFam and bin2chen.

[H-03] ConvexStakingWrapper.sol after shutdown， rewards can be stolen

Submitted by [bin2chen]

After shutdown, checkpoints are stopped, leading to possible theft of rewards.

Proof of Concept

`ConvexStakingWrapper` No more `checkpoints` after `shutdown`, i.e. no updates `reward.reward_integral_for[user]`

```
        function _beforeTokenTransfer(
            address _from,
            address _to,
            uint256
        ) internal override {
    @>      _checkpoint([_from, _to]);
        }

        function _checkpoint(address[2] memory _accounts) internal n
            //if shutdown, no longer checkpoint in case there are pr
    @>      if (isShutdown()) return;

            uint256 supply = _getTotalSupply();
            uint256[2] memory depositedBalance;
            depositedBalance[0] = _getDepositedBalance(_accounts[0])
            depositedBalance[1] = _getDepositedBalance(_accounts[1])

            IRewardStaking(convexPool).getReward(address(this), true

            _claimExtras();

            uint256 rewardCount = rewards.length;
            for (uint256 i = 0; i < rewardCount; i++) {
                _calcRewardIntegral(i, _accounts, depositedBalance, 
            }
        }
```

This would result in, after `shutdown`, being able to steal `rewards` by transferring `tokens` to new users.

Example:
Suppose the current
`reward.reward_integral = 1000`

When a `shutdown` occurs:

1. Alice transfers 100 to the new user, Bob.

Since Bob is the new user and `_beforeTokenTransfer()->_checkpoint()` is not actually executed.
Result:
balanceOf[bob] = 100
reward.reward_integral_for[bob] = 0

2. Bob executes `claimRewards()` to steal the reward.

reward amount = balanceOf[bob] * (reward.reward_integral - reward.reward_integral_for[bob])
= 100 * (1000-0)

3. Bob transfers the balance to other new users, looping steps 1-2 and stealing all rewards.

Recommended Mitigation Steps

Still execute `\_checkpoint`

```
        function _checkpoint(address[2] memory _accounts) internal no
            //if shutdown, no longer checkpoint in case there are pro
-           if (isShutdown()) return;

            uint256 supply = _getTotalSupply();
            uint256[2] memory depositedBalance;
            depositedBalance[0] = _getDepositedBalance(_accounts[0])
            depositedBalance[1] = _getDepositedBalance(_accounts[1])

            IRewardStaking(convexPool).getReward(address(this), true

            _claimExtras();

            uint256 rewardCount = rewards.length;
            for (uint256 i = 0; i < rewardCount; i++) {
                _calcRewardIntegral(i, _accounts, depositedBalance,
            }
        }
```

Assessed type

Context

pmckelvy1 (Reserve) acknowledged

Reserve Mitigated:

> Skip reward claim in `_checkpoint` if shutdown.
> PR: https://github.com/reserve-protocol/protocol/pull/930

Status: Mitigation confirmed. Full details in reports from ronnyx2017, bin2chen and RaymondFam.

Medium Risk Findings (15)

[M-01] Curve Read-only Reentrancy can increase the price of some CurveStableCollateral

Submitted by ronnyx2017*, also found by* bin2chen

If the curve pool of a CurveStableCollateral is a Plain Pool with a native gas token, just like eth/stETH pool:
https://etherscan.io/address/0xdc24316b9ae028f1497c275eb9192a3ea0f67022#code
The price can be manipulated by Curve Read-only Reentrancy.

A example is eth/stETH pool, in its `remove_liquidity` function:

```
    # snippet from remove_liquidity
    CurveToken(lp_token).burnFrom(msg.sender, _amount)
    for i in range(N_COINS):
        value: uint256 = amounts[i] * _amount / total_supply
        if i == 0:
            raw_call(msg.sender, b"", value=value)
        else:
        assert ERC20(self.coins[1]).transfer(msg.sender, value)
```

First, LP tokens are burned. Next, each token is transferred out to the msg.sender. Given that ETH will be the first coin transferred out, token balances and total LP token supply will be inconsistent during the execution of the fallback function.

The `CurveStableCollateral` uses `total underlying token balance value / lp supply` to calculate the lp token price:

```
        (uint192 aumLow, uint192 aumHigh) = totalBalancesValue();

        // {tok}
        uint192 supply = shiftl_toFix(lpToken.totalSupply(), -int8(l
        // We can always assume that the total supply is non-zero

        // {UoA/tok} = {UoA} / {tok}
        low = aumLow.div(supply, FLOOR);
        high = aumHigh.div(supply, CEIL);
```

So the price will be higher than the actual value because the other assets(except eth) are still in the pool but the lp supply has been cut down during the `remove_liquidity` fallback.

tbrent (Reserve) commented via duplicate issue `#14`:

> We have considered this very issue before and have decided as a solution to avoid raw ETH and ERC777's entirely, and *not* try to detect reentrancy in the way described in the article. This is for a few reasons:

1. It cannot be implemented uniformly. `withdraw_admin_fees` is not on all Curve pools, and in particular not on Tricrypto. https://etherscan.io/address/0xd51a44d3fae010294c616388b506acda1bfaae46

2. Raw ETH presents other challenges. Even if we detect reentrancy in the way suggested, the assetRegistry making multiple asset `refresh()` calls means an attacker could gain execution under a *different* asset's refresh() and use that to manipulate refPerTok().

> Also related: Our target unit system does not work well with volatile pools. Each pool would need its own target unit and therefore could not be backed up with any other collateral except identically/distributed pools. We plan to remove `CurveVolatileCollateral` entirely.

tbrent (Reserve) commented via duplicate issue `#14`:

> There is documentation on the website indicating to avoid ERC777 tokens, but this should probably be updated to include forbidding LP tokens that contain raw ETH. Though, it is a bit more complicated than that since some LP tokens offer withdrawal functions that automate the unwrapping of WETH into ETH.

> https://reserve.org/protocol/rtokens/#non-compatible-erc20-assets

pmckelvy1 (Reserve) confirmed via duplicate issue `#14`

Reserve Mitigated:

> Removes `CurveVolatileCollateral`.
> PR: https://github.com/reserve-protocol/protocol/pull/896

Status: Mitigation confirmed. Full details in reports from RaymondFam, ronnyx2017 and bin2chen.

[M-02] `CTokenV3Collateral._underlyingRefPerTok` should use the decimals from underlying Comet

Submitted by sces60107

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CTokenV3Collateral.sol#L56
https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/pl

ugins/assets/compoundv3/CusdcV3Wrapper.sol#L46
https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol#L236

`CTokenV3Collateral._underlyingRefPerTok` uses `erc20Decimals` which is the decimals of `CusdcV3Wrapper`. But it should use the decimals of the underlying Comet.

Proof of Concept

`CTokenV3Collateral.\_underlyingRefPerTok` computes the actual quantity of whole reference units per whole collateral tokens. And it passes `erc20Decimals` to `shiftl_toFix`.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CTokenV3Collateral.sol#L56

```
    function _underlyingRefPerTok() internal view virtual overri
        return shiftl_toFix(ICusdcV3Wrapper(address(erc20)).exch
    }
```

However, the correct decimals should be the decimals of underlying Comet since it is used in `CusdcV3Wrapper.exchangeRate`.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol#L236

```
    function exchangeRate() public view returns (uint256) {
        (uint64 baseSupplyIndex, ) = getUpdatedSupplyIndicies();
        return presentValueSupply(baseSupplyIndex, safe104(10**u
    }
```

Recommended Mitigation Steps

```diff
     function _underlyingRefPerTok() internal view virtual overri
-        return shiftl_toFix(ICusdcV3Wrapper(address(erc20)).excha
+        return shiftl_toFix(ICusdcV3Wrapper(address(erc20)).excha
     }
```

Assessed type

Decimal

tbrent (Reserve) confirmed

pmckelvy1 (Reserve) commented:

> https://github.com/reserve-protocol/protocol/pull/889

Reserve Mitigated:

> Use decimals from underlying Comet.
> PR: https://github.com/reserve-protocol/protocol/pull/889

Status: Mitigation confirmed. Full details in reports from RaymondFam, ronnyx2017 and bin2chen.

[M-03] `RTokenAsset` price estimation accounts for margin of error twice

Submitted by 0xA5DF

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/RTokenAsset.sol#L53-L72
https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/RTokenAsset.sol#L100-L115

`RTokenAsset` estimates the price by multiplying the BU (basket unit) price estimation by the estimation of baskets held (then dividing by total supply). The issue is that both

BU and baskets held account for price margin of error, widening the range of the price more than necessary.

Impact

This would increase the high estimation of the price and decrease the lower estimation. This would impact:

- Setting a lower min price for trading (possibly selling the asset for less than its value)
- Preventing the sale of the asset (`lotLow` falling below the min trade volume)
- Misestimation of the basket range on the 'parent' RToken

Proof of Concept

- Both `tryPrice()` and `lotPrice()` use this method of multiplying basket unit price by basket range then dividing by total supply
- BU price accounts for oracle error
- As for the basket range - whenever one of the collaterals is missing (i.e. less than baskets needed) it estimates the value of anything above the min baskets held, and when doing that it estimates for oracle error as well.

Consider the following scenario:

- We have a basket composed of 1 ETH token and 1 USD token (cUSDCv2)
- cUSDCv2 defaults and the backup token AAVE-USDC kicks in
- Before trading rebalances things we have 0 AAVE-USDC
- This means that we'd be estimating the low price of the ETH we're accounting for margin of error at least twice:
  - Within the `basketRange()` we're dividing the ETH's `low` price by `buPriceHigh`
  - Then we multiply again by `buPriceLow`

(There's also some duplication within the `basketRange()` but that function isn't in scope, what is is scope is the additional margin of error when multiplying by `buPriceLow`).

Recommended Mitigation Steps

I think the best way to mitigate this would be to use a dedicated function to estimate the price, I don’t see an easy way to fix this while using the existing functions.

Assessed type

Other

tbrent (Reserve) commented:

> Currently contemplating switching `BasketHandler.price()/lotPrice()` to return a point estimate, since it is only ever used by `RTokenAsset` and `RecollateralizationLib` to back out a `UoA` value to a `BU` value.

> (or equivalently, using the `basketHandler.price()` midpoint)

tbrent (Reserve) commented:

> @0xA5DF - On further thought I’m not so sure this is a bug, or at least, I don’t think one could do better. Consider the following:

- When an RToken is 100% collateralized (or expects to regain it), `basketRange().top == basketRange().bottom`. But there still needs to be uncertainty associated with the RToken price. It doesn’t make sense for the RToken price estimate to be a single point estimate given there are price uncertainties associated with the backing tokens.
- The behavior you’re describing only occurs when `basketRange()` has a non-zero delta between `top` and `bottom`. The delta exists due to potential clearing prices during the trading that will occur on the way to recollateralization. After all that occurs, there is then an *additional* uncertainty that comes from pricing the tokens that will eventually back the RToken. So it seems right to me to take the oracleError into account twice, for balances that are expected to be traded.

> As for the impact statements, there are a few things I’d point out:

```
- Setting a lower min price for trading (possibly selling the as
- Preventing the sale of the asset (lotLow falling below the min
- Misestimation of the basket range on the 'parent' RToken
```

1. Any RToken sitting in the BackingManager is dissolved as a first step before `rebalance()` trading. RToken will therefore never be bought or sold by the BackingManager, only ever by the RevenueTraders, and RevenueTraders do not pay attention to minTradeVolume.
2. The trading mechanisms are intentionally resilient to under/over-pricing. Batch auctions have good price discovery as long as there is competition, and the dutch auctions will cover the entire distance between the "best price" and "worst price" for the pair, and then some. The impact for dutch auctions would be less precision in the overall clearing price due to larger drops in price per-block. The degree to which it can be said that the asset was sold for less than its true value is thus extremely small, and as implied by point 1 this can only happen for revenue auctions.

0xA5DF (warden) commented:

> So it seems right to me to take the oracleError into account twice, for balances that are expected to be traded.

> I agree there's some sense to it, but:

- The required trading can be a very small percentage of the total basket value, e.g. we have 99 cUSDC and 1 aUSDC and the aUSDC is the one failing. In this case only 1% will be traded while we account for an oracle error for the whole basket.
- Notice that the same thing happens upwards, i.e. we account for the oracle error twice when calculating the `high` price. Do we expect to get more value by trading? we might argue that yes, but I think most cases we lose some value by trading (though I'm not sure what's the impact of the high price being to high)

> Any RToken sitting in the BackingManager is dissolved as a first step before rebalance()

> My understanding was that `RTokenAsset` is for cases when you have one RToken that holds another RToken as an asset, if this isn't the case then I agree this isn't relevant for rebalancing.

> The trading mechanisms are intentionally resilient to under/over-pricing.

> I agree the mechanism will work well for most of the time, but during busy and high gas price periods this might fail and this is when you need the minimum price to kick in.
> Also notice that Dutch trades might have less participants when selling a high volume since it requires to buy the whole batch at once (if I'm not mistaken, I read somewhere in the docs that this is the reason we need `EasyAuction` as well), this is also a case where you need the min price protection mechanism.

tbrent (Reserve) acknowledged and commented:

> Good points. I was ignoring the fact that RTokens may hold other RTokens as assets. In the case of `CurveStableRTokenMetapoolCollateral` , it's also possible for an RToken to hold an LP token for a pool that contains a different RToken as one of its tokens. Currently for example this pool is a collateral token in the RToken hyUSD.

> The point about the uncertainty being applied to the entire basket because of the use of `basketsHeld.bottom` is good as well. If the basket is DISABLED or directly after it is changed, for example, this value would be 0, so the uncertainty would be applied to *all* token balances. This is something we were aware of in the context of a single RToken iteratively recollateralizing (because each step raises `basketsHeld.bottom` , decreasing uncertainty) but it's true that when it comes to one RToken pricing another RToken it seems like it could lead to poor behavior.

> For upwards pricing it feels like less of a concern to me, because `range.top` is bounded at `rToken.basketsNeeded()` .

> It's worth noting though that all this discussion has been in the absence of any RSR stake. In practice all RTokens are overcollateralized by RSR. If the overcollateralization is at least 2%, and the avg oracleError for the collateral tokens is 1%, then ~no double counting occurs because `range.top ~= range.bottom` . Only ETH+ today has such a low overcollateralization; the other 4 RTokens listed on register.app are overcollateralized 7-24%. Still, the protocol should function well when the Distributor is set up with 0% of revenue going to stakers.

> Thoughts on ways this issue could be mitigated? I thought I had an idea but after I looked into it more I don't think it would work.

0xA5DF commented:

> Thoughts on ways this issue could be mitigated?

> Maybe the most simple solution would be to calculate the total value of the assets that the protocol holds (capped to BU price), and then multiply by baskets needed and divide by `totalSupply`.

> I was thinking of modifying `RecollateralizationLibP1.basketRange()` to calculate the price rather than baskets the protocol holds, but I think it'd just be a more complicated way to calculate the above.

tbrent (Reserve) commented:

> Maybe the most simple solution would be to calculate the total value of the assets that the protocol holds (capped to BU price), and then multiply by baskets needed and divide by totalSupply.

> The issue with an approach like this is that it's agnostic of where we are in the collateralization process. Balances that are disjoint with the current basket are treated the same as balances that are overlapping. This really comes down to the question of when and where to apply `maxTradeSlippage` and subtract out `minTradeVolume`, which is what the current `basketRange()` implementation aims to do.

tbrent (Reserve) disagreed with severity and commented:

> We've discussed internally and where we're coming down is that we think this issue should be acknowledged but that it is a Medium and not a High. We want to acknowledge the issue because while we were aware of the double-counting of oracleError in the context of a single RToken pricing itself, we hadn't considered it in the context of a parent-child relationship, and in that case it is importantly different. However, it seems more like a Medium because the trading mechanisms are resilient to mild mispricing. The expected downside outcome would be a trade occuring via `DutchTrade` and the block-by-block price dropping faster than necessary, possibly resulting in more slippage, but this would likely be very small and on the order of ~0.1%, and only for the impacted balance held in the child RToken.

cccz (judge) decreased severity to Medium

Reserve Mitigated:

> Acknowledged and documented.
> PR: https://github.com/reserve-protocol/protocol/pull/916

Status: Mitigation confirmed. Full details in reports from bin2chen, RaymondFam and ronnyx2017.

[M-04] Possible rounding during the reward calculation

Submitted by auditor0517

Some rewards might be locked inside the contract due to the rounding loss.

Proof of Concept

`_claimAndSyncRewards()` claimed the rewards from the staking contract and tracks `rewardsPerShare` with the current supply.

```
    function _claimAndSyncRewards() internal virtual {
        uint256 _totalSupply = totalSupply();
        if (_totalSupply == 0) {
            return;
        }
        _claimAssetRewards();
        uint256 balanceAfterClaimingRewards = rewardToken.balanc

        uint256 _rewardsPerShare = rewardsPerShare;
        uint256 _previousBalance = lastRewardBalance;

        if (balanceAfterClaimingRewards > _previousBalance) {
            uint256 delta = balanceAfterClaimingRewards - _previ
            // {qRewards/share} += {qRewards} * {qShare/share} /
            _rewardsPerShare += (delta * one) / _totalSupply; //
        }
        lastRewardBalance = balanceAfterClaimingRewards;
        rewardsPerShare = _rewardsPerShare;
    }
```

It uses one as a multiplier and from this setting we know it has the same decimals as `underlying` (thus `totalSupply`).

My concern is `_claimAndSyncRewards()` is called for each deposit/transfer/withdraw in _beforeTokenTransfer() and it will make the rounding problem more serious.

1. Let's consider underlyingDecimals = 18. `totalSupply = 10**6 with 18 decimals`, `rewardToken` has 6 decimals. And total rewards for 1 year are `1M rewardToken` for `1M totalSupply`.
2. With the above settings, `_claimAndSyncRewards()` might be called every 1 min due to the frequent user actions.
3. Then expected rewards for 1 min are `1000000 / 365 / 24 / 60 = 1.9 rewardToken = 1900000 wei`.
4. During the division, it will be `1900000 * 10**18 / (1000000 * 10**18) = 1`.

So users would lose almost 50% of rewards due to the rounding loss and these rewards will be locked inside the contract.

Recommended Mitigation Steps

I think there would be 2 mitigations.

1. Use a bigger multiplier.
2. Keep the remainders and use them next time in `_claimAndSyncRewards()` like this.

```
    if (balanceAfterClaimingRewards > _previousBalance) {
        uint256 delta = balanceAfterClaimingRewards - _previousB
        uint256 deltaPerShare = (delta * one) / _totalSupply; //

        // decrease balanceAfterClaimingRewards so remainders ca
        balanceAfterClaimingRewards = _previousBalance + deltaPe

        _rewardsPerShare += deltaPerShare;
    }
    lastRewardBalance = balanceAfterClaimingRewards;
```

Assessed type

Math

tbrent (Reserve) confirmed and commented:

> Mitigation option `#2` seems quite good.

Reserve Mitigated:

> Roll over remainder to next call.
> PR: https://github.com/reserve-protocol/protocol/pull/896

Status: Mitigation confirmed. Full details in reports from ronnyx2017, RaymondFam and bin2chen.

[M-05] Permanent funds lock in `StargateRewardableWrapper`

Submitted by auditor0517

The staked funds might be locked because the deposit/withdraw/transfer logic reverts.

Proof of Concept

In `StargateRewardableWrapper`, `_claimAssetRewards()` claims the accumulated rewards from the staking contract and it's called during every deposit/withdraw/transfer in _beforeTokenTransfer() and _claimAndSyncRewards().

```
    function _claimAssetRewards() internal override {
        stakingContract.deposit(poolId, 0);
    }
```

And in the stargate staking contract, deposit() calls updatePool() inside the function.

```
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply = pool.lpToken.balanceOf(address(this))
        if (lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock,
        uint256 stargateReward = multiplier.mul(stargatePerBlock

        pool.accStargatePerShare = pool.accStargatePerShare.add(
        pool.lastRewardBlock = block.number;
    }

    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        ...
    }
```

The problem is `updatePool()` reverts when `totalAllocPoint == 0` and this value can be changed by stargate admin using [set()](#).

So user funds might be locked like the below.

1. The stargate staking contract had one pool and `totalAllocPoint = 10`.
2. In `StargateRewardableWrapper`, some users staked their funds using [deposit()](#).
3. After that, that pool was removed by the stargate admin due to an unexpected reason. So the admin called [set(0, 0)](#) to reset the pool. Then `totalAllocPoint = 0` now. In the stargate contract, it's not so critical because this contract has [emergencyWithdraw()](#) to rescue funds without caring about rewards. Normal users can withdraw their funds using this function.
4. But in `StargateRewardableWrapper`, there is no logic to be used under the emergency and deposit/withdraw won't work because `_claimAssetRewards()` reverts in [updatePool()](#) due to 0 division.

Recommended Mitigation Steps

We should implement a logic for an emergency in `StargateRewardableWrapper`.

During the emergency, `_claimAssetRewards()` should return 0 without interacting with the staking contract and we should use `stakingContract.emergencyWithdraw()` to rescue the funds.

Assessed type

Error

[cccz (judge) decreased severity to Medium and commented](#):

> External requirement with specific owner behavior.

[tbrent (Reserve) confirmed](#)

[Reserve Mitigated](#):

> Add call to `emergencyWithdraw`.
> PR: https://github.com/reserve-protocol/protocol/pull/896

Status: Mitigation confirmed. Full details in reports from ronnyx2017, bin2chen and RaymondFam.

[M-06] CurveStableMetapoolCollateral.tryPrice returns a huge but valid high price when the price oracle of pairedToken is timeout

Submitted by ronnyx2017

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/CurveStableMetapoolCollateral.sol#L83-L86
https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/CurveStableCollateral.sol#L74-L98

The CurveStableMetapoolCollateral is intended for 2-fiattoken stable metapools. The metapoolToken coin0 is pairedToken and the coin1 is lpToken, e.g. 3CRV. And the `config.chainlinkFeed` should be set for paired token.

Impact

The CurveStableMetapoolCollateral.price() high price will be about `FIX_MAX / metapoolToken.totalSupply()` when the price oracle of pairedToken is timeout. It is significantly more than the actual price. It will lead to unexpected pricing in the rewards trade and rebalance auctions. Furthermore, I think an attacker can trigger this bug proactively by out of gas, which can bypass the empty error message check because of the different call stack depth. But I have not verified the idea due to lack of time. So the issue here only details the high price caused by external factor, for example oracle timeout. Hope to add it under this issue if I have any other progress. Thanks.

Proof of Concept

In the `CurveStableMetapoolCollateral.tryPrice` function, the pairedToken price is from `tryPairedPrice` function by the following codes:

```
    uint192 lowPaired;
    uint192 highPaired = FIX_MAX;
    try this.tryPairedPrice() returns (uint192 lowPaired_, uint192 h
        lowPaired = lowPaired_;
        highPaired = highPaired_;
    } catch {}

    function tryPairedPrice() public view virtual returns (uint192 l
        uint192 p = chainlinkFeed.price(oracleTimeout); // {UoA/pair
        uint192 delta = p.mul(oracleError, CEIL);
        return (p - delta, p + delta);
    }
```

So if the chainlinkFeed is offline(oracle timeout), the tryPairedPrice will throw an error which is caught by the empty catch block, and the price of pairedToken will be (0, FIX_MAX).

And then the function `_metapoolBalancesValue` will use these prices to get the total UoA of the metapool. The following codes are how it uses the price of pairedToken:

```
    aumLow += lowPaired.mul(pairedBal, FLOOR);

    // Add-in high part carefully
    uint192 toAdd = highPaired.safeMul(pairedBal, CEIL);
    if (aumHigh + uint256(toAdd) >= FIX_MAX) {
        aumHigh = FIX_MAX;
    } else {
        aumHigh += toAdd;
    }
```

The `aumLow` has already included the UoA of LpToken, so it is non-zero. And the highPaired price now is FIX_MAX, which will mul the paired token balance by `Fixed.safeMul`. We can find the Fixed lib has handled overflow safely:

```
    function safeMul(
        uint192 a,
        uint192 b,
        RoundingMode rounding
    ) internal pure returns (uint192) {
        ...
        if (a == FIX_MAX || b == FIX_MAX) return FIX_MAX;
```

So the `aumHigh` from the `_metapoolBalancesValue` function will be FIX_MAX. The final prices are calculated by:

```
    low = aumLow.div(supply, FLOOR);
    high = aumHigh.div(supply, CEIL);
```

`supply` is the `metapoolToken.totalSupply()`. So if the supply is > 1 token, the `Fixed.div` won’t revert. And the high price will be a huge but valid value < FIX_MAX.

Recommended Mitigation Steps

Don’t try catch the `this.tryPairedPrice()` in the `CurveStableMetapoolCollateral.tryPrice`, if it failed, just let the whole tryPrice function revert, the caller, for example refresh(), can catch the error.

Assessed type

Context

[tbrent (Reserve) confirmed](#)

[Reserve Mitigated](#):

> Enforce (`0, FIX_MAX`) as "unpriced" during oracle timeout.
> PR: https://github.com/reserve-protocol/protocol/pull/917

Status: Mitigation confirmed. Full details in reports from ronnyx2017, RaymondFam and bin2chen.

[M-07] The Asset.lotPrice doubles the oracle timeout in the worst case

Submitted by ronnyx2017

When the `tryPrice()` function revert, for example oracle timeout, the `Asset.lotPrice` will use a decayed historical value:

```
    uint48 delta = uint48(block.timestamp) - lastSave; // {s}
    if (delta <= oracleTimeout) {
        lotLow = savedLowPrice;
        lotHigh = savedHighPrice;
    } else if (delta >= oracleTimeout + priceTimeout) {
        return (0, 0); // no price after full timeout
    } else {
```

And the delta time is from the last price saved time. If the delta time is greater than oracle timeout, historical price starts decaying.

But the last price might be saved at the last second of the last oracle timeout period. So the `Asset.lotPrice` will double the oracle timeout in the worst case.

Impact

The `Asset.lotPrice` will double the oracle timeout in the worst case. When the rewards need to be sold or basket is rebalancing, if the price oracle is offline temporarily, the `Asset.lotPrice` will use the last saved price in max two oracle timeout before the historical value starts to decay. It increases the sale/buy price of the asset.

Proof of Concept

The `lastSave` is updated in the `refresh()` function, and it's set to the current `block.timestamp` instead of the `updateTime` from the chainlink feed:

```
    function refresh() public virtual override {
        try this.tryPrice() returns (uint192 low, uint192 high, uint
            if (high < FIX_MAX) {
                savedLowPrice = low;
                savedHighPrice = high;
                lastSave = uint48(block.timestamp);
```

But in the `OracleLib`, the oracle time is checked for the delta time of `block.timestamp - updateTime`:

```
    uint48 secondsSince = uint48(block.timestamp - updateTime);
    if (secondsSince > timeout) revert StalePrice();
```

So if the last oracle feed updateTime is `block.timestamp - priceTimeout`, the timeout check will be passed and lastSave will be updated to block.timestamp. And the lotPrice will start to decay from `lastSave + priceTimeout`. However when it starts, it's been 2 * priceTimeout since the last oracle price update.

Recommended Mitigation Steps

Starts lotPrice decay immediately or updated the `lastSave` to `updateTime` instead of `block.timestamp`.

Assessed type

Context

tbrent (Reserve) disputed and commented:

> This issue was known and was discussed internally. Unfortunately this occurred in the private copy of the repo that the devs use to coordinate while C4 audits are ongoing. I've attached a screenshot of the discussion, though it is up to C4 how to treat this ultimately. We could probably provide repo access to a member of the C4 team if asked.

> Screenshot 2023-08-08 at 4 11 33 PM

> We decided not to pursue this direction as it introduced a large number of changes, and it seems acceptable to have the worst-case behavior of using 100% of the last saved price for up to one oracleTimeout too long.

cccz (judge) commented:

> Agree that sponsors not address it.
> And this issue will be considered as medium risk under the C4 criteria.
> `> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

[M-08] User can't redeem from RToken based on CurveStableRTokenMetapoolCollateral when any underlying collateral of paired RToken's price oracle is offline(timeout)

Submitted by ronnyx2017

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/CurveStableRTokenMetapoolCollateral.sol#L46-L54
https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/CurveStableCollateral.sol#L119-L121
https://github.com/reserve-

protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/CurveStableMetapoolCollateral.sol#L122-L138

The CurveStableMetapoolCollateral is intended for 2-fiattoken stable metapools that involve RTokens, such as eUSD-fraxBP. The metapoolToken coin0 is pairedToken, which is also a RToken, and the coin1 is lpToken, e.g. 3CRV. And the `CurveStableRTokenMetapoolCollateral.tryPairedPrice` uses `RTokenAsset.price()` as the oracle to get the pairedToken price:

```
    function tryPairedPrice()
        ...
        returns (uint192 lowPaired, uint192 highPaired)
    {
        return pairedAssetRegistry.toAsset(pairedToken).price();
    }
```

Impact

Users can't redeem from RToken when any underlying collateral of paired RToken's price oracle is offline(timeout). It can lead to a serious run/depeg on the RToken.

Proof of Concept

First I submitted another issue named "RTokenAsset price oracle can return a huge but valid high price when any underlying collateral's price oracle timeout". It's the premise for this issue. Because this issue is located in different collateral codes, I split them into two issues.

The conclusion from the pre issue:

> *If there is any underlying collateral's price oracle reverts, for example oracle timeout, the* `RTokenAsset.price` *will return a valid but untrue (low, high) price range, which can be described as* `low = true_price * A1` *and* `high = FIX_MAX * A2` *, A1 is* `bh.quantity(oracle_revert_coll) / all quantity for a BU` *and A2 is the* `BasketRange.top / RToken totalSupply` *.*

Back to the `CurveStableRTokenMetapoolCollateral`. There are two cases that will revert in the super class `CurveStableCollateral.refresh()`.

The `CurveStableRTokenMetapoolCollateral.tryPairedPrice` function gets low/high price from `paired RTokenAsset.price()`. So when any underlying collateral's price oracle of paired RTokenAsset reverts, the max high price will be FIX_MAX and the low price is non-zero.

1. If the high price is FIX_MAX, the assert for low price will revert:

```
        if (high < FIX_MAX) {
            savedLowPrice = low;
            savedHighPrice = high;
            lastSave = uint48(block.timestamp);
        } else {
            // must be unpriced
            // untested:
            //      validated in other plugins, cost to test here is
            assert(low == 0);
        }
```

2. And if high price is There is a little smaller than FIX_MAX, the `_anyDepeggedOutsidePool` check in the refresh function will revert.

```
        if (low == 0 || _anyDepeggedInPool() || _anyDepeggedOutsideP
            markStatus(CollateralStatus.IFFY);
        }
```

And the `CurveStableMetapoolCollateral` overrides it:

```
         function _anyDepeggedOutsidePool() internal view virtual ov
            try this.tryPairedPrice() returns (uint192 low, uint192 l
                // {UoA/tok} = {UoA/tok} + {UoA/tok}
                uint192 mid = (low + high) / 2;

                // If the price is below the default-threshold price
                // uint192(+/-) is the same as Fix.plus/minus
                if (mid < pairedTokenPegBottom || mid > pairedTokenPe
            }
```

So the `uint192 mid = (low + high) / 2;` will revert because of uint192 overflow. The `CurveStableRTokenMetapoolCollateral.refresh()` will revert without any catch.

Because RToken.redeemTo and redeemCustom need to call `assetRegistry.refresh();` at the beginning, it will revert directly.

Recommended Mitigation Steps

The Fix.plus can't handle the uint192 overflow error. Try to override `_anyDepeggedOutsidePool` for `CurveStableRTokenMetapoolCollateral` as:

```
        unchecked {
            uint192 mid = (high - low) / 2 + low;
        }
```

The assert `assert(low <= high)` in the RTokenAsset.tryPrice has already protected everything.

Assessed type

DoS

[cccz (judge) decreased severity to Medium and commented](#):

> External requirement with oracle errors.

[pmckelvy1 (Reserve) confirmed](#)

[Reserve Mitigated](#):

> Unpriced on oracle timeout.
> PR: https://github.com/reserve-protocol/protocol/pull/917

Status: Mitigation confirmed. Full details in reports from [ronnyx2017](#), [RaymondFam](#) and [bin2chen](#) .

[M-09] RTokenAsset price oracle can return a huge but valid high price when any underlying collateral's price oracle timeout

Submitted by ronnyx2017

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/RTokenAsset.sol#L163-L175
https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/RTokenAsset.sol#L53-L69
https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/p1/BasketHandler.sol#L329-L351

The RTokenAsset is an implementation of interface `IRTokenOracle` to work as a oracle price feed for the little RToken. RTokenAsset implements the `latestPrice` function to get the oracle price and saved time from the `cachedOracleData`, which is updated by `_updateCachedPrice` function:

```
    function _updateCachedPrice() internal {
        (uint192 low, uint192 high) = price();

        require(low != 0 && high != FIX_MAX, "invalid price");

        cachedOracleData = CachedOracleData(
            (low + high) / 2,
            block.timestamp,
            basketHandler.nonce(),
            backingManager.tradesOpen(),
            backingManager.tradesNonce()
        );
    }
```

The `_updateCachedPrice` gets the low and high prices from `price()`, and updates the oracle price to `(low + high) / 2`. And it checks `low != 0 && high != FIX_MAX`.

The `RTokenAsset.price` just uses the return of `tryPrice` as the low price and high price, if `tryPrice` reverts, it will return `(0, FIX_MAX)`, which is an invalid price range for the oracle price check above. But if there is any underlying collateral's price oracle reverts, for example oracle timeout, the `RTokenAsset.price` will return a valid but untrue (low, high) price range, which can be described as `low = true_price * A1` and `high = FIX_MAX * A2`, A1 is `bh.quantity(oracle_revert_coll) / all quantity for a BU` and A2 is the `BasketRange.top / RToken totalSupply`.

Impact

The RToken oracle price will be about `FIX_MAX / 2` when any underlying collateral's price oracle is timeout. It is significantly more than the actual price. It will lead to a distortion in the price of collateral associated with the RToken, for example `CurveStableRTokenMetapoolCollateral`:

```
        pairedAssetRegistry = IRToken(address(pairedToken)).main().a

        function tryPairedPrice()
        ...
        {
            return pairedAssetRegistry.toAsset(pairedToken).price();
        }
```

Proof of Concept

`RToken.tryPrice` gets the BU (low, high) price from `basketHandler.price()` first. `BasketHandler._price(false)` core logic:

```
    for (uint256 i = 0; i < len; ++i) {
        uint192 qty = quantity(basket.erc20s[i]);

        (uint192 lowP, uint192 highP) = assetRegistry.toAsset(basket

        low256 += qty.safeMul(lowP, RoundingMode.FLOOR);

        if (high256 < FIX_MAX) {
            if (highP == FIX_MAX) {
                high256 = FIX_MAX;
            } else {
                high256 += qty.safeMul(highP, RoundingMode.CEIL);
            }
        }
    }
```

And the `IAsset.price()` should not revert. If the price oracle of the asset reverts, it just returns `(0,FIX_MAX)`. In this case, the branch will enter `high256 += qty.safeMul(highP, RoundingMode.CEIL);` first. And it won't revert for overflow because the Fixed.safeMul will return FIX_MAX directly if any param is FIX_MAX:

```
    function safeMul(
        ...
    ) internal pure returns (uint192) {
        if (a == FIX_MAX || b == FIX_MAX) return FIX_MAX;
```

So the high price is `FIX_MAX`, and the low price is reduced according to the share of qty.

Return to the `RToken.tryPrice`, the following codes uses `basketRange()` to calculate the low and high price for BU:

```
    BasketRange memory range = basketRange(); // {BU}

    // {UoA/tok} = {BU} * {UoA/BU} / {tok}
    low = range.bottom.mulDiv(lowBUPrice, supply, FLOOR);
    high = range.top.mulDiv(highBUPrice, supply, CEIL);
```

And the only thing has to be proofed is `range.top.mulDiv(highBUPrice, supply, CEIL)` should not revert for overflow in unit192. Now `highBUPrice = FIX_MAX`, according to the `Fixed.mulDiv`, if `range.top <= supply` it won't overflow. And for passing the check in the `RToken._updateCachedPrice()`, the high price should be lower than `FIX_MAX`. So it needs to ensure `range.top < supply`.

The max value of range.top is basketsNeeded which is defined in `RecollateralizationLibP1.basketRange(ctx, reg)`:

```
    if (range.top > basketsNeeded) range.top = basketsNeeded;
```

And the basketsNeeded:RToken supply is 1:1 at the beginning. If the RToken has experienced a haircut or the RToken is undercollateralized at present, the basketsNeeded can be lower than RToken supply.

Recommended Mitigation Steps

Add a BU price valid check in the `RToken.tryPrice`:

```
    function tryPrice() external view virtual returns (uint192 low, 
        (uint192 lowBUPrice, uint192 highBUPrice) = basketHandler.pr
        require(lowBUPrice != 0 && highBUPrice != FIX_MAX, "invalid 
```

Assessed type

Context

cccz (judge) decreased severity to Medium and commented:

> External requirement with oracle errors.

pmckelvy1 (Reserve) confirmed

Reserve Mitigated:

> Enforce (`0, FIX_MAX`) as "unpriced" during oracle timeout.
> PR: https://github.com/reserve-protocol/protocol/pull/917

Status: Mitigation confirmed. Full details in reports from ronnyx2017, RaymondFam and bin2chen.

[M-10] `Asset.lotPrice` only uses `oracleTimeout` to determine if the price is stale.

Submitted by sces60107

`OracleTimeout` is the number of seconds until an oracle value becomes invalid. It is set in the constructor of `Asset`. And `Asset.lotPrice` uses `OracleTimeout` to determine if the saved price is stale. However, `OracleTimeout` may not be the correct source to determine if the price is stale. `Asset.lotPrice` may return the incorrect price.

Proof of Concept

`OracleTimeout` is set in the constructor of `Asset`.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/Asset.sol#L61

```
    constructor(
        uint48 priceTimeout_,
        AggregatorV3Interface chainlinkFeed_,
        uint192 oracleError_,
        IERC20Metadata erc20_,
        uint192 maxTradeVolume_,
        uint48 oracleTimeout_
    ) {
        …
        oracleTimeout = oracleTimeout_;
    }
```

`Asset.lotPrice` use `oracleTimeout` to determine if the saved price is in good standing.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/Asset.sol#L140

```
    function lotPrice() external view virtual returns (uint192 lo
        try this.tryPrice() returns (uint192 low, uint192 high,
            // if the price feed is still functioning, use that
            lotLow = low;
            lotHigh = high;
        } catch (bytes memory errData) {
            // see: docs/solidity-style.md#Catching-Empty-Data
            if (errData.length == 0) revert(); // solhint-disabl
            // if the price feed is broken, use a decayed histor

            uint48 delta = uint48(block.timestamp) - lastSave; /
            if (delta <= oracleTimeout) {
                lotLow = savedLowPrice;
                lotHigh = savedHighPrice;
            } else if (delta >= oracleTimeout + priceTimeout) {
                return (0, 0); // no price after full timeout
            } else {
                // oracleTimeout <= delta <= oracleTimeout + pri

                // {1} = {s} / {s}
                uint192 lotMultiplier = divuu(oracleTimeout + pr

                // {UoA/tok} = {UoA/tok} * {1}
                lotLow = savedLowPrice.mul(lotMultiplier);
                lotHigh = savedHighPrice.mul(lotMultiplier);
            }
        }
        assert(lotLow <= lotHigh);
    }
```

However, `oracleTimeout` may not be the accurate source to determine if the saved price is stale. The following examples shows that using only `oracleTimeout` is vulnerable.

1. `NonFiatCollateral.tryPrice` leverages two price feeds to calculate the price. These two feeds have different timeouts.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/NonFiatCollateral.sol#L52

```
    function tryPrice()
        external
        view
        override
        returns (
            uint192 low,
            uint192 high,
            uint192 pegPrice
        )
    {
        pegPrice = chainlinkFeed.price(oracleTimeout); // {targe

        // Assumption: {ref/tok} = 1; inherit from `Appreciating
        // {UoA/tok} = {UoA/target} * {target/ref} * {ref/tok} (
        uint192 p = targetUnitChainlinkFeed.price(targetUnitOrac

        // this oracleError is already the combined total oracle
        uint192 err = p.mul(oracleError, CEIL);

        low = p - err;
        high = p + err;
        // assert(low <= high); obviously true just by inspectio
    }
```

If `targetUnitChainlinkFeed` is malfunctioning and `targetUnitOracleTimeout` is smaller than `oracleTimeout`, `lotPrice()` should not return saved price when `delta > targetUnitOracleTimeout`. However, `lotPrice()` only considers `oracleTimeout`. It could return the incorrect price when `targetUnitChainlinkFeed` is malfunctioning.

2. To calculate the price, `CurveStableCollateral.tryPrice` calls `PoolToken.totalBalancesValue`. And `PoolToken.totalBalancesValue` calls `PoolToken.tokenPrice`. `PoolToken.tokenPrice` uses multiple feeds to calculate the price. And they could have different timeouts. None of them are used in `lotPrice()`.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/CurveStableCollateral.sol#L57
https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/PoolTokens.sol#L287
https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/PoolTokens.sol#L235

```
    function tryPrice()
        external
        view
        virtual
        override
        returns (
            uint192 low,
            uint192 high,
            uint192
        )
    {
        // {UoA}
        (uint192 aumLow, uint192 aumHigh) = totalBalancesValue()

        // {tok}
        uint192 supply = shiftl_toFix(lpToken.totalSupply(), -in
        // We can always assume that the total supply is non-zer

        // {UoA/tok} = {UoA} / {tok}
        low = aumLow.div(supply, FLOOR);
        high = aumHigh.div(supply, CEIL);
        assert(low <= high); // not obviously true just by inspe

        return (low, high, 0);
    }


    function totalBalancesValue() internal view returns (uint192
        for (uint8 i = 0; i < nTokens; ++i) {
            IERC20Metadata token = getToken(i);
            uint192 balance = shiftl_toFix(curvePool.balances(i)
            (uint192 lowP, uint192 highP) = tokenPrice(i);

            low += balance.mul(lowP, FLOOR);
            high += balance.mul(highP, CEIL);
        }
    }

    function tokenPrice(uint8 index) public view returns (uint19
        ...

        if (index == 0) {
            x = _t0feed0.price(_t0timeout0);
            xErr = _t0error0;
            if (address(_t0feed1) != address(0)) {
                y = _t0feed1.price(_t0timeout1);
                yErr = _t0error1;
            }
        ...
        return toRange(x, y, xErr, yErr);
    }
```

We can also find out that `oracleTimeout` is unused. But `lotPrice()` still uses it to determine if the saved price is valid. This case is worse than the first case.
https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/CurveStableCollateral.sol#L28

```
    /// @dev config Unused members: chainlinkFeed, oracleError, 
    /// @dev config.erc20 should be a RewardableERC20
    constructor(
        CollateralConfig memory config,
        uint192 revenueHiding,
        PTConfiguration memory ptConfig
    ) AppreciatingFiatCollateral(config, revenueHiding) PoolToke
        require(config.defaultThreshold > 0, "defaultThreshold z
    }
```

Recommended Mitigation Steps

Since collaterals have various implementations of price feed. `Asset.lotPrice` could be modified like:

```
    function lotPrice() external view virtual returns (uint192 l
        …
-           if (delta <= oracleTimeout) {
+           if (delta <= actualOracleTimeout()) {
                lotLow = savedLowPrice;
                lotHigh = savedHighPrice;
        …
    }

+   function actualOracleTimeout() public view virtual returns (
+       return oracleTimeout;
+   }
```

Then, collaterals can override `actualOracleTimeout` to reflect the correct oracle timeout.

Assessed type

Error

cccz (judge) decreased severity to Medium and commented:

> External requirement with oracle errors.

tbrent (Reserve) acknowledged

pmckelvy1 (Reserve) commented:

> As documented in `Asset.sol` here:
>
> `oracleTimeout_` is also used as the timeout value in `lotPrice()`; should be highest of all assets 'oracleTimeout in a collateral if there are multiple oracles.

[M-11] StaticATokenLM transfer missing _updateRewards

Submitted by bin2chen

Transfer missing `_updateRewards()`, resulting in the loss of `from`'s reward.

Proof of Concept

`StaticATokenLM` contains the rewards mechanism, when the balance changes, the global `_accRewardsPerToken` needs to be updated first to calculate the user's `rewardsAccrued` more accurately.

Example: `mint()/burn()` both call `_updateRewards()` to update `_accRewardsPerToken`.

```
    function _deposit(
        address depositor,
        address recipient,
        uint256 amount,
        uint16 referralCode,
        bool fromUnderlying
    ) internal returns (uint256) {
        require(recipient != address(0), StaticATokenErrors.INVA
@>          _updateRewards();

...

        _mint(recipient, amountToMint);

        return amountToMint;
    }


    function _withdraw(
        address owner,
        address recipient,
        uint256 staticAmount,
        uint256 dynamicAmount,
        bool toUnderlying
    ) internal returns (uint256, uint256) {
...
@>          _updateRewards();

...
@>          _burn(owner, amountToBurn);

...
    }
```

When `transfer()/transerFrom()` , the balance is also modified, but without calling `_updateRewards()` first. The result is that if the user transfers the balance, the difference in rewards accrued by `from` is transferred to `to` along with it. This doesn't make sense for `from` .

```
        function _beforeTokenTransfer(
            address from,
            address to,
            uint256
        ) internal override {
            if (address(INCENTIVES_CONTROLLER) == address(0)) {
                return;
            }
            if (from != address(0)) {
                _updateUser(from);
            }
            if (to != address(0)) {
                _updateUser(to);
            }
        }
```

Recommended Mitigation Steps

`_beforeTokenTransfer` first trigger `_updateRewards()`.

```
        function _beforeTokenTransfer(
            address from,
            address to,
            uint256
        ) internal override {
            if (address(INCENTIVES_CONTROLLER) == address(0)) {
                return;
            }
+           _updateRewards();
            if (from != address(0)) {
                _updateUser(from);
            }
            if (to != address(0)) {
                _updateUser(to);
            }
        }
```

Assessed type

Context

tbrent (Reserve) commented:

> We will be reaching out to the Aave team to understand more about this. It seems there are multiple places in StaticATokenLM where reward steps are missing, and there may be reasons why.

julianmrodri (Reserve) commented:

> We will mark this issue as Sponsor Acknowledged. It is true the situation described by the warden and that's the behavior we observe. However we will not be implementing any change in the code (besides adding some comments) for the following reasons:

- We do not expect rewards in Aave V2 to come back.
- We checked with Aave and we believe the original reason for building it this way still holds, and that is for gas purposes. Even if for some reason rewards on AAve V2 come back the cost of updating the user rewards on every transfer outweighs the rewards that may be left "uncollected" after a `transfer` operation. It is important to remark that any `deposit` or `withdraw` done to the contract plus any call to `collectRewards..`, and any claim of rewards from the Reserve protocol, would setup the correct balances. So while it is true that transfers may in some cases not transfer rewards we expect this to only be slightly off.

> To clarify this issue for users we will add a comment to the wrapper contract `StaticATokenLM` to clarify the situation with how rewards are handled on transfer.

tbrent (Reserve) acknowledged

[M-12] `_claimRewardsOnBehalf()` User's rewards may be lost

Submitted by bin2chen, *also found by* carlitox477

Incorrect determination of maximum rewards, which may lead to loss of user rewards.

Proof of Concept

`_claimRewardsOnBehalf()` for users to retrieve rewards:

```
        function _claimRewardsOnBehalf(
            address onBehalfOf,
            address receiver,
            bool forceUpdate
        ) internal {
            if (forceUpdate) {
                _collectAndUpdateRewards();
            }

            uint256 balance = balanceOf(onBehalfOf);
            uint256 reward = _getClaimableRewards(onBehalfOf, balanc
            uint256 totBal = REWARD_TOKEN.balanceOf(address(this));

@>          if (reward > totBal) {
@>              reward = totBal;
@>          }
```

```
        if (reward > 0) {
@>          _unclaimedRewards[onBehalfOf] = 0;
            _updateUserSnapshotRewardsPerToken(onBehalfOf);
            REWARD_TOKEN.safeTransfer(receiver, reward);
        }
    }
```

From the code above, we can see that if the contract balance is not enough, it will only use the contract balance and set the unclaimed rewards to 0: `_unclaimedRewards[user]=0`.

But using the current contract's balance is inaccurate, `REWARD_TOKEN` may still be stored in `INCENTIVES_CONTROLLER`.

`_updateRewards()` and `_updateUser()`, are just calculations, they don't transfer `REWARD_TOKEN` to the current contract, but `_unclaimedRewards[user]` is always accumulating.

1. `_updateRewards()` not transferable `REWARD_TOKEN`.

```
        function _updateRewards() internal {
    ...
            if (block.number > _lastRewardBlock) {
    ...

                address[] memory assets = new address[](1);
                assets[0] = address(ATOKEN);

    @>          uint256 freshRewards = INCENTIVES_CONTROLLER.getI
                uint256 lifetimeRewards = _lifetimeRewardsClaime
                uint256 rewardsAccrued = lifetimeRewards.sub(_li

    @>          _accRewardsPerToken = _accRewardsPerToken.add(
                    (rewardsAccrued).rayDivNoRounding(supply.wad
                );
                _lifetimeRewards = lifetimeRewards;
            }
        }
```

2. But `_unclaimedRewards[user]` always accumulating.

```
    function _updateUser(address user) internal {
        uint256 balance = balanceOf(user);
        if (balance > 0) {
            uint256 pending = _getPendingRewards(user, balance, 
@>          _unclaimedRewards[user] = _unclaimedRewards[user].ad
        }
        _updateUserSnapshotRewardsPerToken(user);
    }
```

This way if `_unclaimedRewards(forceUpdate=false)` is executed, it does not trigger the transfer of `REWARD_TOKEN` to the current contract. This makes it possible that `_unclaimedRewards[user] > REWARD_TOKEN.balanceOf(address(this))`. According to the `_claimedRewardsOnBehalf()` current code, the extra value is lost.

It is recommended that `if (reward > totBal)` be executed only if `forceUpdate=true`, to avoid losing user rewards.

Recommended Mitigation Steps

```
    function _claimRewardsOnBehalf(
        address onBehalfOf,
        address receiver,
        bool forceUpdate
    ) internal {
        if (forceUpdate) {
            _collectAndUpdateRewards();
        }

        uint256 balance = balanceOf(onBehalfOf);
        uint256 reward = _getClaimableRewards(onBehalfOf, balanc
        uint256 totBal = REWARD_TOKEN.balanceOf(address(this));

-       if (reward > totBal) {
+       if (forceUpdate && reward > totBal) {
            reward = totBal;
        }
        if (reward > 0) {
            _unclaimedRewards[onBehalfOf] = 0;
            _updateUserSnapshotRewardsPerToken(onBehalfOf);
            REWARD_TOKEN.safeTransfer(receiver, reward);
        }
        }
```

Assessed type

Context

[cccz (judge) decreased severity to Medium]

[tbrent (Reserve) commented]:

> See comment on issue 12.

[julianmrodri (Reserve) commented]:

> After a thorough review we can confirm this is not an issue. This is the way it should work and that's the reason why there is a forceUpdate param. When forceUpdate == true, then you will always have the latest rewards to claim and the updated balance.

> When is set to false, it will only distribute the rewards that were previously collected (the ones available in the contract). It is correct there might be additional rewards to be collected, but that can easily be done with another call to the same function using the forceUpdate == true.

> There are no rewards "lost" in the process, no fix needs to be implemented. Even though `unclaimedRewards` is set to zero, then it will be populated with all the `pending` rewards again so the amount will be ok.

> Moreover, the suggested fix would brick the function most of the time (as usually rewards are bigger than balance because it includes uncollected but pending rewards), and in that case it would attempt to transfer rewards not available in the contract. The check of just sending the balance in those cases is required.

[tbrent (Reserve) disputed]

[cccz (judge) commented]:

> @bin2chen - please take a look.
> It seems that since `_getPendingRewards` has a false parameter, `_unclaimedRewards` does not accumulate unclaimed rewards in the controller, so the rewards are not lost.

```
    function _updateUser(address user) internal {
        uint256 balance = balanceOf(user);
        if (balance > 0) {
            uint256 pending = _getPendingRewards(user, balance, :
            _unclaimedRewards[user] = _unclaimedRewards[user].ad(
        }
        _updateUserSnapshotRewardsPerToken(user);
    }
```

bin2chen (warden) commented:

> @cccz - `getPendingRewards(fresh = true)`. It doesn't matter if `fresh` is `true` or `false`, because this can only be used to calculate the latest global `accRewardsPerToken`.

> Since `_updateRewards()` must be executed before `_updateUser (user)` is executed to ensure that `accRewardsPerToken` is up-to-date, it does not matter whether `fresh` is true.

> But the message above
> `Even though unclaimedRewards is set to zero, then it will be populated with all the pending rewards again so the amount will be ok.`

> It confuses me a bit, I might need to take another look. I need to familiarize myself with this project again to see if I missed something.

cccz (judge) commented:

> `_getClaimableRewards` returns `_unclaimedRewards + pendingRewards`, that is, `reward = _unclaimedRewards + pendingRewards`, so just setting

> `_unclaimedRewards` to 0 will not decrease pendingRewards, which may be somewhat helpful.

```
            uint256 reward = _getClaimableRewards(onBehalfOf, balanc
            uint256 totBal = REWARD_TOKEN.balanceOf(address(this));

            if (reward > totBal) {
                reward = totBal;
            }
            if (reward > 0) {
                _unclaimedRewards[onBehalfOf] = 0;
                _updateUserSnapshotRewardsPerToken(onBehalfOf);
                REWARD_TOKEN.safeTransfer(receiver, reward);
            }
    ...
        function _getClaimableRewards(
            address user,
            uint256 balance,
            bool fresh
        ) internal view returns (uint256) {
            uint256 reward = _unclaimedRewards[user].add(_getPending
            return reward.rayToWadNoRounding();
        }
```

bin2chen (warden) commented:

> @cccz - `pendingRewards` is assumed to be 0.
> But `_unclaimedRewards[user]` has a value, the point is that the value in there is not in the current contract, it's in `INCENTIVES_CONTROLLER`. If it's cleared, it's gone.
> I think I need to take another look.

bin2chen (warden) commented:

> @cccz - I'll keep my original point. Please help me see if I'm missing something. Thanks.

> The current implementation only moves rewards to the current contract if `_collectAndUpdateRewards()` is executed.

> `_updateRewards()` and `_updateUser()` are not triggered.

> But `_unclaimedRewards[user]` is accumulated.
> `_accRewardsPerToken` and `_userSnapshotRewardsPerToken[user]` keeps getting bigger.

> So that if no one has called `_collectAndUpdateRewards()` (i.e. forceUpdate=false is not called).

> This way the rewards balance in the contract will always be zero.

> After `_claimRewardsOnBehalf(forceUpdate=false)`.
> The user doesn't get any rewards, but `_unclaimedRewards[user]` is cleared to 0 and can't be refilled (note that it's not pendingRewards, assuming that pendingRewards is 0).

> This way the rewards are lost.

cccz (judge) commented:

> Need review from sponsors. @julianmrodri

bin2chen (warden) commented:

> Here's a test case to look at.
> Note: The balance of the current contract described above cannot be 0, it needs to be a little bit.

> Add to StaticATokenLM.test.ts

```
    it('test_lost', async () => {
      const amountToDeposit = utils.parseEther('5')

      // Just preparation
      await waitForTx(await weth.deposit({ value: amountToDeposi
      await waitForTx(
        await weth.approve(staticAToken.address, amountToDeposit
      )

      // Depositing
      await waitForTx(
        await staticAToken.deposit(userSigner._address, amountTo
      )
```

```
        await advanceTime(1);
        //***** need small reward balace
        await staticAToken.collectAndUpdateRewards()
        const staticATokenBalanceFirst = await stkAave.balanceOf(s
        await advanceTime(60 * 60 * 24)

        // Depositing
        await waitForTx(
          await staticAToken.deposit(userSigner._address, amountTo
        )

        const beforeRewardBalance = await stkAave.balanceOf(userSi
        const pendingRewardsBefore = await staticAToken.getClaimab
        console.log("user Reward Balance(Before):",beforeRewardBal

        // user claim forceUpdate = false
        await waitForTx(await staticAToken.connect(userSigner).cla

        const afterRewardBalance = await stkAave.balanceOf(userSig
        const pendingRewardsAfter = await staticAToken.getClaimabl
        console.log("user Reward Balance(After):",afterRewardBalan


        const pendingRewardsDecline = pendingRewardsBefore.toNumbe
        const getRewards= afterRewardBalance.toNumber() - beforeRe
        console.log("user pendingRewardsBefore:",pendingRewardsBef
        console.log("user pendingRewardsAfter:",pendingRewardsAfte
        const staticATokenBalanceAfter = await stkAave.balanceOf(s
        console.log("staticAToken Balance (before):",staticATokenB
        console.log("staticAToken Balance (After):",staticATokenBa
        console.log("user lost:",pendingRewardsDecline - getReward



      })
```

```
`$` yarn test:plugins:integration --grep "test_lost"
```

```
  StaticATokenLM: aToken wrapper with static balances and liquid
Duplicate definition of RewardsClaimed (RewardsClaimed(address,a
    Rewards - Small checks
user Reward Balance(Before): BigNumber { value: "0" }
user Reward Balance(After): BigNumber { value: "34497547939" }
user pendingRewardsBefore: BigNumber { value: "1490345817336159"
user pendingRewardsAfter: BigNumber { value: "34497293495" }
staticAToken Balance (before): BigNumber { value: "34497547939"
staticAToken Balance (After): BigNumber { value: "0" }
user lost: 1490276822494725
      ✓ test_lost (947ms)


  1 passing (24s)
```

[julianmrodri (Reserve) commented]():

> Thanks for the example. I'll review and let you know.

[julianmrodri (Reserve) commented]():

> @cccz & @bin2chen - Ok, the issue exists I can confirm now. This is a tricky one, nice catch. Found out that this was built this way on purpose. The idea is to allow the user to "sacrifice" some rewards for gas savings. You can see it in some of the tests, with comments like this one:

```
expect(pendingRewards5).to.be.eq(0) // User "sacrifice" excess r
```

> We will discuss today what action we are taking with this issue.

> In any case, the suggested mitigation does not address fully the issue, and causes the contract to fail under normal operations (simply try to run our test suite with that change). I believe we should probably address the main issue that the `_unclaimed` variable is set to 0, instead of to the rewards still pending to be collected.

> What do you think about the function working this way? I ran a simple check and seems to address it at least for the example you provided. This mitigation was suggested on the other ticket linked here, it had some rounding issues but overall is the same.

```
  function _claimRewardsOnBehalf(
        address onBehalfOf,
        address receiver,
        bool forceUpdate
    ) internal {
        if (forceUpdate) {
            _collectAndUpdateRewards();
        }

        uint256 balance = balanceOf(onBehalfOf);
        uint256 reward = _getClaimableRewards(onBehalfOf, balanc
        uint256 totBal = REWARD_TOKEN.balanceOf(address(this));

        if (reward == 0) {
            return;
       }

        if (reward > totBal) {
            reward = totBal;
            _unclaimedRewards[onBehalfOf] -= reward.wadToRay();
        } else {
            _unclaimedRewards[onBehalfOf] = 0;
        }

        _updateUserSnapshotRewardsPerToken(onBehalfOf);
        REWARD_TOKEN.safeTransfer(receiver, reward);
    }
```

> Thanks for taking a look!

bin2chen (warden) commented:

> @julianmrodri - This one still has problems.

1. If there is a `pendingReward` may `underflow`

   For example:

   balance = 10

   _unclaimedRewards[user]=9

   pendingRewards = 2

> `_unclaimedRewards[onBehalfOf] -= reward.wadToRay();` will underflow

2. Finally executed `_updateUserSnapshotRewardsPerToken(onBehalfOf);`

   Then we need to accumulate `pendingRewards` to `_unclaimedRewards[user]`.

> Personally, I feel that if we don't want to revert, try this:

```
    function _claimRewardsOnBehalf(
        address onBehalfOf,
        address receiver,
        bool forceUpdate
    ) internal {
        if (forceUpdate) {
            _collectAndUpdateRewards();
        }

        uint256 balance = balanceOf(onBehalfOf);
        uint256 reward = _getClaimableRewards(onBehalfOf, balanc
        uint256 totBal = REWARD_TOKEN.balanceOf(address(this));

        if (reward == 0) {
            return;
       }

        if (reward > totBal) {
+           // Insufficient balance resulting in no transfers out
+           _unclaimedRewards[onBehalfOf] = (reward -totBal).wad
            reward = totBal;
-           _unclaimedRewards[onBehalfOf] -= reward.wadToRay();
        } else {
            _unclaimedRewards[onBehalfOf] = 0;
        }

        _updateUserSnapshotRewardsPerToken(onBehalfOf);
        REWARD_TOKEN.safeTransfer(receiver, reward);
    }
```

> This may still have this prompt.
> `expect(pendingRewards5).to.be.eq(0) // User "sacrifice" excess rewards to save on gas-costs`

> But I feel that this use case should be changed. It is okay to sacrifice a little. If it is a lot, it is still necessary to prevent the user from executing it.

pmckelvy1 (sponsor) acknowledged

julianmrodri (Reserve) commented:

> @cccz @bin2chen - We had a group call and we decided to ACKNOWLEDGE the issue but we will not make code changes, just add a comment in the contract explaining this risk of losing rewards if you call it with the `false` parameter.

> The reasons are:

- We do not expect Aave V2 Rewards to come back in practice.
- Our protocol is not exposed to this issue. Because in our code we always claim with the parameter set to `true`, and nobody can claim on behalf of the protocol with the `false` parameter, the Protocol will always get the latest rewards when claiming.
- It is up to the user to call this with true or false. They might see value in calling it with false and sacrificing some rewards, which was the original reason it was built this way. But they always have the option to call it with the true parameter which will behave normally. So we consider it more like an option they have and not that much of a bug. We also do not expect people holding this wrapper token besides using it in our protocol.

> However, we acknowledge and value the finding which was spot on and allowed us to understand the wrapper in more detail. Thanks for that!

[M-13] Lack of protection when caling `CusdcV3Wrapper._withdraw`

Submitted by [RaymondFam]

When unwrapping the `wComet` to its rebasing `comet`, users with an equivalent amount of `wComet` invoking `CusdcV3Wrapper._withdraw` at around the same time could end up having different percentage gains because `comet` is not linearly rebasing.

Moreover, the rate-determining `getUpdatedSupplyIndicies()` is an internal view function inaccessible to the users unless they take the trouble creating a contract to inherit CusdcV3Wrapper.sol. So most users making partial withdrawals will have no clue whether or not this is the best time to unwrap. This is because the public view function [underlyingBalanceOf] is only directly informational when `amount` has been entered as `type(uint256).max`.

Proof of Concept

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/pl

ugins/assets/compoundv3/CusdcV3Wrapper.sol#L134-L170

```
    function _withdraw(
        address operator,
        address src,
        address dst,
        uint256 amount
    ) internal {
        if (!hasPermission(src, operator)) revert Unauthorized()
        // {Comet}
        uint256 srcBalUnderlying = underlyingBalanceOf(src);
        if (srcBalUnderlying < amount) amount = srcBalUnderlying
        if (amount == 0) revert BadAmount();

        underlyingComet.accrueAccount(address(this));
        underlyingComet.accrueAccount(src);

        uint256 srcBalPre = balanceOf(src);
        CometInterface.UserBasic memory wrappedBasic = underlyin
        int104 wrapperPrePrinc = wrappedBasic.principal;

        // conservative rounding in favor of the wrapper
        IERC20(address(underlyingComet)).safeTransfer(dst, (amou

        wrappedBasic = underlyingComet.userBasic(address(this));
        int104 wrapperPostPrinc = wrappedBasic.principal;

        // safe to cast because principal can't go negative, wra
        uint256 burnAmt = uint256(uint104(wrapperPrePrinc - wrap
        // occasionally comet will withdraw 1-10 wei more than w
        // this is ok because 9 times out of 10 we are rounding
        // safe because we have already capped the comet withdra
        // untested:
        //      difficult to trigger, depends on comet rules reg
        if (srcBalPre <= burnAmt) burnAmt = srcBalPre;

        accrueAccountRewards(src);
        _burn(src, safe104(burnAmt));
    }
```

As can be seen in the code block of function `_withdraw` above, `underlyingBalanceOf(src)` is first invoked.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol#L225-L231

```
    function underlyingBalanceOf(address account) public view re
        uint256 balance = balanceOf(account);
        if (balance == 0) {
            return 0;
        }
        return convertStaticToDynamic(safe104(balance));
    }
```

Next, function `convertStaticToDynamic` is invoked.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol#L241-L244

```
    function convertStaticToDynamic(uint104 amount) public view 
        (uint64 baseSupplyIndex, ) = getUpdatedSupplyIndicies();
        return presentValueSupply(baseSupplyIndex, amount);
    }
```

And next, function `getUpdatedSupplyIndicies` is invoked. As can be seen in its code logic, the returned value of `baseSupplyIndex_` is determined by the changing `supplyRate`.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol#L298-L313

```
    function getUpdatedSupplyIndicies() internal view returns (u
        TotalsBasic memory totals = underlyingComet.totalsBasic(
        uint40 timeDelta = uint40(block.timestamp) - totals.last
        uint64 baseSupplyIndex_ = totals.baseSupplyIndex;
        uint64 trackingSupplyIndex_ = totals.trackingSupplyIndex
        if (timeDelta > 0) {
            uint256 baseTrackingSupplySpeed = underlyingComet.ba
            uint256 utilization = underlyingComet.getUtilization
            uint256 supplyRate = underlyingComet.getSupplyRate(u
            baseSupplyIndex_ += safe64(mulFactor(baseSupplyIndex_
            trackingSupplyIndex_ += safe64(
                divBaseWei(baseTrackingSupplySpeed * timeDelta,
            );
        }
        return (baseSupplyIndex_, trackingSupplyIndex_);
    }
```

The returned value of `baseSupplyIndex` is then inputted into function `principalValueSupply` where the lower the value of `baseSupplyIndex`, the higher the `principalValueSupply` or simply put, the lesser the `burn` amount.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CometHelpers.sol#L29-L35

```
    function principalValueSupply(uint64 baseSupplyIndex_, uint2
        internal
        pure
        returns (uint104)
    {
        return safe104((presentValue_ * BASE_INDEX_SCALE) / base
    }
```

Recommended Mitigation Steps

Consider implementing slippage protection on `CusdcV3Wrapper._withdraw` so that users could opt for the minimum amount of `comet` to receive or the maximum amount of `wComet` to burn.

Assessed type

Timing

tbrent (Reserve) acknowledged

pmckelvy1 (Reserve) commented:

> Users can always use `convertStaticToDynamic` and `convertDynamicToStatic` to get the exchange rates as they both use `getUpdatedSupplyIndicies()`. The issue being flagged here (rebase rate is dynamic) is inherent to the comet itself (and pretty much any rebasing token for that matter), and not something the wrapper needs to be concerned about.

[M-14] Lack of protection when withdrawing Static Atoken

Submitted by [RaymondFam]()

The Aave plugin is associated with [an ever-increasing exchange rate](). The earlier a user wraps the AToken, the more Static Atoken will be minted and understandably no slippage protection is needed.

However, since the rate is not linearly increasing, withdrawing the Static Atoken (following RToken redemption) at the wrong time could mean a difference in terms of the amount of AToken redeemed. The rate could be in a transient mode of non-increasing or barely increasing and then a significant surge. Users with an equivalent amount of Static AToken making such calls at around the same time could end up having different percentage gains.

Although the user could always deposit and wrap the AToken again, it's not going to help if the wrapping were to encounter a sudden surge (bad timing again) thereby thwarting the intended purpose.

Proof of Concept

[https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/aave/StaticATokenLM.sol#L338-L355]()

```
        uint256 userBalance = balanceOf(owner);

        uint256 amountToWithdraw;
        uint256 amountToBurn;

        uint256 currentRate = rate();
        if (staticAmount > 0) {
            amountToBurn = (staticAmount > userBalance) ? userBa
            amountToWithdraw = _staticToDynamicAmount(amountToBu
        } else {
            uint256 dynamicUserBalance = _staticToDynamicAmount(
            amountToWithdraw = (dynamicAmount > dynamicUserBalan
                ? dynamicUserBalance
                : dynamicAmount;
            amountToBurn = _dynamicToStaticAmount(amountToWithdr
        }

        _burn(owner, amountToBurn);
```

As can be seen in the code block of function `_withdraw` above, choosing `staticAmount > 0` will have a lesser amount of AToken to withdraw when the `currentRate` is stagnant.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/aave/StaticATokenLM.sol#L297-L299

```
    function _staticToDynamicAmount(uint256 amount, uint256 rate_
        return amount.rayMul(rate_);
    }
```

Similarly, choosing `dynamicAmount > 0` will have a higher than expected amount of Static Atoken to burn.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/aave/StaticATokenLM.sol#L293-L295

```
    function _dynamicToStaticAmount(uint256 amount, uint256 rate_
        return amount.rayDiv(rate_);
    }
```

Recommended Mitigation Steps

Consider implementing slippage protection on `StaticATokenLM._withdraw` so that users could opt for the minimum amount of AToken to receive or the maximum amount of Static Atoken to burn.

Assessed type

Timing

tbrent (Reserve) acknowledged

julianmrodri (Reserve) commented:

> Hi, as mentioned before we acknowledge the existence of this issue.

> But on the other hand we will not implement fixes or changes. We believe it is the responsibility of the user to decide when to wrap/unwrap these tokens and these interactions are in general outside of the protocol behavior.

> In addition to this the rate is accesible for the user to make that decision, and we don't expect these rates to increase abruptly for this wrapper, so in reality we might be adding a feature that will probably not be used in practice.

> It is important to remark this is something that exists in any wrapper for rebasing tokens we use, whether it is our own, or developed by other protocol teams. And generally we don't see implemented in those wrappers slippage protection or a feature like the one suggested here.

[M-15] Potential Loss of Rewards During Token Transfers in StaticATokenLM.sol

Submitted by RaymondFam

This issue could lead to a permanent loss of rewards for the transferer of the token. During the token transfer process, the `_beforeTokenTransfer` function updates rewards for both the sender and the receiver. However, due to the specific call order and the behavior of the `_updateUser` function and the `_getPendingRewards` function, some rewards may not be accurately accounted for.

The crux of the problem lies in the fact that the `_getPendingRewards` function, when called with the `fresh` parameter set to `false`, may not account for all the latest rewards from the `INCENTIVES_CONTROLLER`. As a result, the `_updateUserSnapshotRewardsPerToken` function, which relies on the output of the `_getPendingRewards` function, could end up missing out on some rewards. This could be detrimental to the token sender, especially the `Backing Manager` whenever `RToken` is redeemed. Apparently, most users having wrapped their `AToken`, would automatically use it to issue `RToken` as part of the basket range of backing collaterals and be minimally impacted. But it would have the Reserve Protocol collectively affected depending on the frequency and volume of RToken redemption and the time elapsed since `_updateRewards()` or `_collectAndUpdateRewards()` was last called. The same impact shall also apply when making token transfers to successful auction bidders.

Proof of Concept

As denoted in the function NatSpec below, function `_beforeTokenTransfer` updates rewards for senders and receivers in a `transfer`.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/aave/StaticATokenLM.sol#L367-L386

```
    /**
      * @notice Updates rewards for senders and receiver in a tra
      * @param from The address of the sender of tokens
      * @param to The address of the receiver of tokens
      */
     function _beforeTokenTransfer(
         address from,
         address to,
         uint256
     ) internal override {
         if (address(INCENTIVES_CONTROLLER) == address(0)) {
             return;
         }
         if (from != address(0)) {
             _updateUser(from);
         }
         if (to != address(0)) {
             _updateUser(to);
         }
```

```
}
```

When function `_updateUser` is respectively invoked, `_getPendingRewards(user, balance, false)` is called.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/aave/StaticATokenLM.sol#L543-L550

```
/**
 * @notice Adding the pending rewards to the unclaimed for s
 * @param user The address of the user to update
 */
function _updateUser(address user) internal {
    uint256 balance = balanceOf(user);
    if (balance > 0) {
        uint256 pending = _getPendingRewards(user, balance, 
        _unclaimedRewards[user] = _unclaimedRewards[user].ad
    }
    _updateUserSnapshotRewardsPerToken(user);
}
```

However, because the third parameter has been entered `false`, the third if block of function `_getPendingRewards` is skipped.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/aave/StaticATokenLM.sol#L552-L591

```
/**
 * @notice Compute the pending in RAY (rounded down). Pendin
 * @param user The user to compute for
 * @param balance The balance of the user
 * @param fresh Flag to account for rewards not claimed by c
 * @return The amount of pending rewards in RAY
 */
function _getPendingRewards(
    address user,
    uint256 balance,
    bool fresh
) internal view returns (uint256) {
    if (address(INCENTIVES_CONTROLLER) == address(0)) {
        return 0;
    }

    if (balance == 0) {
        return 0;
    }

    uint256 rayBalance = balance.wadToRay();

    uint256 supply = totalSupply();
    uint256 accRewardsPerToken = _accRewardsPerToken;

    if (supply != 0 && fresh) {
        address[] memory assets = new address[](1);
        assets[0] = address(ATOKEN);

        uint256 freshReward = INCENTIVES_CONTROLLER.getReward
        uint256 lifetimeRewards = _lifetimeRewardsClaimed.add
        uint256 rewardsAccrued = lifetimeRewards.sub(_lifetim
        accRewardsPerToken = accRewardsPerToken.add(
            (rewardsAccrued).rayDivNoRounding(supply.wadToRa
        );
    }

    return
        rayBalance.rayMulNoRounding(accRewardsPerToken.sub(_
}
```

Hence, `accRewardsPerToken` may not be updated with the latest rewards from `INCENTIVES_CONTROLLER`. Consequently, `_updateUserSnapshotRewardsPerToken(user)` will miss out on claiming some rewards and is a loss to the StaticAtoken transferrer.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/aave/StaticATokenLM.sol#L531-L537

```
    /**
     * @notice Update the rewardDebt for a user with balance as l
     * @param user The user to update
     */
    function _updateUserSnapshotRewardsPerToken(address user) in
        _userSnapshotRewardsPerToken[user] = _accRewardsPerToken
    }
```

Ironically, this works out as a zero-sum game where the loss of the transferrer is a gain to the transferee. But most assuredly, the Backing Manager is going to end up incurring more losses than gains in this regard.

Recommended Mitigation Steps

Consider introducing an additional call to update the state of rewards before any token transfer occurs. Specifically, within the `_beforeTokenTransfer` function, invoking `_updateRewards` like it has been implemented in `StaticATokenLM._deposit` and `StaticATokenLM._withdraw` before updating the user balances would have the issues resolved.

This method would not force an immediate claim of rewards, but rather ensure the internal accounting of rewards is up-to-date before the transfer. By doing so, the state of rewards for each user would be accurate and ensure no loss or premature gain of rewards during token transfers.

Assessed type

Token-Transfer

cccz (judge) decreased severity to Medium

tbrent (Reserve) commented:

> See comment on issue 12.

julianmrodri (Reserve) commented:

> We will mark this issue as Sponsor Acknowledged. It is true the situation described by the warden and that's the behavior we observe. However we will not be implementing any change in the code (besides adding some comments) for the following reasons:

- We do not expect rewards in Aave V2 to come back.

- We checked with Aave and we believe the original reason for building it this way still holds, and that is for gas purposes. Even if for some reason rewards on AAve V2 come back the cost of updating the user rewards on every transfer outweighs the rewards that may be left "uncollected" after a `transfer` operation. It is important to remark that any `deposit` or `withdraw` done to the contract plus any call to `collectRewards..`, and any claim of rewards from the Reserve protocol, would setup the correct balances. So while it is true that transfers may in some cases not transfer rewards we expect this to only be slightly off.

> To clarify this issue for users we will add a comment to the wrapper contract `StaticATokenLM` to clarify the situation with how rewards are handled on transfer.

tbrent (Reserve) acknowledged

Low Risk and Non-Critical Issues

For this audit, 6 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **auditor0517** received the top score from the judge.

The following wardens also submitted reports: ronnyx2017, bin2chen, RaymondFam, carlitox477, and 0xA5DF.

[L-01] Unsafe max approve

It's not recommended to approve `type(uint256).max` for safety. We should approve a relevant amount every time.

- https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/aave/StaticATokenLM.sol#L105

```
ASSET.safeApprove(address(pool), type(uint256).max);
```

- https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/stargate/StargateRewardableWrapper.sol#L39

```
pool_.approve(address(stakingContract_), type(uint256).max);
```

[L-02] Possible reentrancy

Users might manipulate `wrapperPostPrinc` inside the transfer hook. With the current `underlyingComet` token, there is no impact as it doesn't have any hook but it's recommended to add a `nonReentrant` modifier to `_deposit()/_withdraw()`.

- https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol#L96

```
IERC20(address(underlyingComet)).safeTransferFrom(src, address(t
```

- https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol#L154

```
IERC20(address(underlyingComet)).safeTransfer(dst, (amount / 10)
```

[L-03] Unsafe downcasting

`feeds[i].length` might be downcasted wrongly when it's greater than `type(uint8).max`. `maxFeedsLength()` might pass the requirement when `config.feeds` has many elements(like 256).

- https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/PoolTokens.sol#L330

```
maxLength = uint8(Math.max(maxLength, feeds[i].length));
```

[L-04] Reverts on 0 transfer

It deposits 0 amount to the staking contract to claim rewards but it might revert during the 0 transfer. There is no problem with the current `lpToken` but good to keep in mind.

- https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/stargate/StargateRewardableWrapper.sol#L48

```
stakingContract.deposit(poolId, 0);
```

[L-05] Wrong comment

- https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/Asset.sol#L146

```
// oracleTimeout <= delta <= oracleTimeout + priceTimeout ======
```

- https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/WrappedERC20.sol#L298

```
     * - when `from` is zero, `amount` tokens will be minted for
     * - when `to` is zero, `amount` of ``from``'s tokens will b
```

[L-06] Unsafe permission

`WrappedERC20` uses a permission mechanism and operators can have an infinite allowance once approved. It might be inconvenient/dangerous for some users.

- https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol#L84

```
    function _deposit(
        address operator,
        address src,
        address dst,
        uint256 amount
    ) internal {
        if (!hasPermission(src, operator)) revert Unauthorized()
        ...
    }
```

- https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol#L140

```
    function _withdraw(
        address operator,
        address src,
        address dst,
        uint256 amount
    ) internal {
        if (!hasPermission(src, operator)) revert Unauthorized()
        ...
    }
```

[L-07] Typical first depositor issue in `RewardableERC4626Vault`

It's recommended to follow the instructions.

- https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/erc20/RewardableERC4626Vault.sol#L20

```
abstract contract RewardableERC4626Vault is ERC4626, RewardableE
```

[N-01] Typo

`_accumuatedRewards` => `_accumulatedRewards`

- https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/erc20/RewardableERC20.sol#L57

```
uint256 _accumuatedRewards = accumulatedRewards[account];
```

[N-02] Needless accrue for src

We don't need to accrue for `src` because we don't use any information of `src` and that info will be accrued during the underlyingComet transfer.

- https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol#L91

```
        underlyingComet.accrueAccount(address(this));
        underlyingComet.accrueAccount(src); //@audit needless accrue

        CometInterface.UserBasic memory wrappedBasic = underlyingCom
        int104 wrapperPrePrinc = wrappedBasic.principal;

        IERC20(address(underlyingComet)).safeTransferFrom(src, addre
```

Gas Optimizations

For this audit, 2 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **RaymondFam** received the top score from the judge.

The following wardens also submitted reports: carlitox477.

[G-01] Immutable over constant

The use of constant `keccak` variables results in extra hashing whenever the variable is used, increasing gas costs relative to just storing the output hash. Changing to immutable will only perform hashing on contract deployment which will save gas.

Here are some of the instances entailed.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/aave/StaticATokenLM.sol#L44-L60

```
        bytes public constant EIP712_REVISION = bytes("1");
        bytes32 internal constant EIP712_DOMAIN =
            keccak256(
                "EIP712Domain(string name,string version,uint256 cha
            );
        bytes32 public constant PERMIT_TYPEHASH =
            keccak256(
                "Permit(address owner,address spender,uint256 value,
            );
        bytes32 public constant METADEPOSIT_TYPEHASH =
            keccak256(
                "Deposit(address depositor,address recipient,uint256
            );
        bytes32 public constant METAWITHDRAWAL_TYPEHASH =
            keccak256(
                "Withdraw(address owner,address recipient,uint256 st
            );
```

[G-02] Unreachable code lines

The following else block in the function logic of `refresh()` is never reachable considering `tryPrice()` does not have (0, FIX_MAX) catered for. Any upriced data would have been sent to the catch block. Other than Asset.sol, similarly wasted logic is also exhibited in `AppreciatingFiatCollateral.refresh`, `FiatCollateral.refresh`, `CTokenV3Collateral.refresh`, `CurveStableCollateral.refresh`, `StargatePoolFiatCollateral.refresh`.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/Asset.sol#L86-L106

```
        /// Should not revert
        /// Refresh saved prices
        function refresh() public virtual override {
```

```
        try this.tryPrice() returns (uint192 low, uint192 high,
            // {UoA/tok}, {UoA/tok}
            // (0, 0) is a valid price; (0, FIX_MAX) is unpriced

            // Save prices if priced
            if (high < FIX_MAX) {
                savedLowPrice = low;
                savedHighPrice = high;
                lastSave = uint48(block.timestamp);
            } else {
                // must be unpriced
                assert(low == 0);
            }
        } catch (bytes memory errData) {
            // see: docs/solidity-style.md#Catching-Empty-Data
            if (errData.length == 0) revert(); // solhint-disabl
        }
    }
```

[G-03] Redundant `return` in the `try` clause

This `try` clause has already returned `low` and `high` and is returning the same things again in its nested logic, which is inexpedient and unnecessary. Similar behavior is also found in `Asset.price`.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/RTokenAsset.sol#L86-L94

```
    function price() public view virtual returns (uint192, uint1
        try this.tryPrice() returns (uint192 low, uint192 high)
-            return (low, high);
        } catch (bytes memory errData) {
            // see: docs/solidity-style.md#Catching-Empty-Data
            if (errData.length == 0) revert(); // solhint-disabl
            return (0, FIX_MAX);
        }
    }
```

[G-04] Unneeded import in `RTokenAsset.sol`

`IRToken.sol` has already been imported by `IMain.sol`, making the former an unneeded import.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/RTokenAsset.sol#L5-L6

```
 import "../../interfaces/IMain.sol";
- import "../../interfaces/IRToken.sol";
```

[G-05] Cached variables not efficiently used

In `StaticATokenLM._updateRewards`, the state variable `_lifetimeRewardsClaimed` is doubly used in the following code logic when `rewardsAccrued` could simply/equally be assigned `freshRewards.wadToRay()`. This extra gas incurring behaviour is also exhibited in `StaticATokenLM._collectAndUpdateRewards`, and `StaticATokenLM._getPendingRewards`.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/aave/StaticATokenLM.sol#L406-L408

```
            uint256 freshRewards = INCENTIVES_CONTROLLER.getRewa
            uint256 lifetimeRewards = _lifetimeRewardsClaimed.add
-            uint256 rewardsAccrued = lifetimeRewards.sub(_lifet
+            uint256 rewardsAccrued = freshRewards.wadToRay();
```

[G-06] Identical for loop check

The if block in the constructor of PoolTokens.sol entails identical if block checks that may be moved outside the loop to save gas.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/PoolTokens.sol#L123-L130

```
        IERC20Metadata[] memory tokens = new IERC20Metadata[](nT

+        if (config.poolType == CurvePoolType.Plain) revert("inv

        for (uint8 i = 0; i < nTokens; ++i) {
-            if (config.poolType == CurvePoolType.Plain) {
                tokens[i] = IERC20Metadata(curvePool.coins(i));
-            } else {
-                revert("invalid poolType");
-            }
        }
```

[G-07] Unneeded ternary logic, booleans, and second condition checks

In the constructor of PoolTokens.sol, the second condition of the following require statement mandates that at least one feed is associated with each token.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/PoolTokens.sol#L106-L109

```
        require(
            config.feeds.length == config.nTokens && minFeedsLen
            "each token needs at least 1 price feed"
        );
```

Additionally, the following code lines signify that a minimum of 2 tokens will be associated with the Curve base pool. Otherwise, if `nTokens` is less than 2 or 1, assigning `token0` and/or `token1` will revert due to accessing out-of-bound array elements.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/PoolTokens.sol#L132-L135

```
        token0 = tokens[0];
        token1 = tokens[1];
        token2 = (nTokens > 2) ? tokens[2] : IERC20Metadata(addr
        token3 = (nTokens > 3) ? tokens[3] : IERC20Metadata(addr
```

Under this context, the following ternary logic along with the use of boolean `more` is therefore deemed unnecessary.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/PoolTokens.sol#L143-L154

```
        // token0
-        bool more = config.feeds[0].length > 0;
        // untestable:
        //     more will always be true based on previous feeds
-        _t0feed0 = more ? config.feeds[0][0] : AggregatorV3Inte
+        _t0feed0 = config.feeds[0][0];
        _t0timeout0 = more && config.oracleTimeouts[0].length >
        _t0error0 = more && config.oracleErrors[0].length > 0 ?
-        if (more) {
            require(address(_t0feed0) != address(0), "t0feed0 em
            require(_t0timeout0 > 0, "t0timeout0 zero");
            require(_t0error0 < FIX_ONE, "t0error0 too large");
-        }
```

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/PoolTokens.sol#L166-L177

```
        // token1
        // untestable:
        //     more will always be true based on previous feeds
-        more = config.feeds[1].length > 0;
-        _t1feed0 = more ? config.feeds[1][0] : AggregatorV3Inte
+        _t1feed0 = config.feeds[1][0];
        _t1timeout0 = more && config.oracleTimeouts[1].length >
        _t1error0 = more && config.oracleErrors[1].length > 0 ?
-        if (more) {
            require(address(_t1feed0) != address(0), "t1feed0 em
            require(_t1timeout0 > 0, "t1timeout0 zero");
            require(_t1error0 < FIX_ONE, "t1error0 too large");
-           }
```

Similarly, the following second conditional checks are also not needed.

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/PoolTokens.sol#L189-L190

```
            // token2
-            more = config.feeds.length > 2 && config.feeds[2].lengt
+            more = config.feeds.length > 2;
```

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/curve/PoolTokens.sol#L210-L211

```
            // token3
-            more = config.feeds.length > 3 && config.feeds[3].lengt
+            more = config.feeds.length > 3;
```

[G-08] Use of named returns for local variables saves gas

You can have further advantages in terms of gas cost by simply using named return values as temporary local variables.

For instance, the code block below may be refactored as follows:

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol#L225

```
-     function underlyingBalanceOf(address account) public view r
+     function underlyingBalanceOf(address account) public view r
          uint256 balance = balanceOf(account);
          if (balance == 0) {
              return 0;
          }
```

```diff
-          return convertStaticToDynamic(safe104(balance));
+          _balance = convertStaticToDynamic(safe104(balance));
      }
```

[G-09] `+=` and `-=` cost more gas

`+=` and `-=` generally cost 22 more gas than writing out the assigned equation explicitly. The amount of gas wasted can be quite sizable when repeatedly operated in a loop.

For instance, the `+=` instance below may be refactored as follows:

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol#L307

```diff
-              baseSupplyIndex_ += safe64(mulFactor(baseSupplyInde
+              baseSupplyIndex_ = baseSupplyIndex_ + safe64(mulFac
```

[G-10] Function order affects gas consumption

The order of function will also have an impact on gas consumption. Because in smart contracts, there is a difference in the order of the functions. Each position will have an extra 22 gas. The order is dependent on method ID. So, if you rename the frequently accessed function to more early method ID, you can save gas cost. Please visit the following site for further information:

https://medium.com/joyso/solidity-how-does-function-name-affect-gas-consumption-in-smart-contract-47d270d8ac92

[G-11] Activate the optimizer

Before deploying your contract, activate the optimizer when compiling using `solc --optimize --bin sourceFile.sol`. By default, the optimizer will optimize the contract assuming it is called 200 times across its lifetime. If you want the initial contract deployment to be cheaper and the later function executions to be more expensive, set it to `--optimize-runs=1`. Conversely, if you expect many transactions and do not care for higher deployment cost and output size, set `--optimize-runs` to a high number.

```
module.exports = {
solidity: {
version: "0.8.19",
settings: {
  optimizer: {
    enabled: true,
    runs: 1000,
  },
},
},
};
```

Please visit the following site for further information:

https://docs.soliditylang.org/en/v0.5.4/using-the-compiler.html#using-the-commandline-compiler

Here's one example of instance on opcode comparison that delineates the gas saving mechanism:

```
for !=0 before optimization
PUSH1 0x00
DUP2
EQ
ISZERO
PUSH1 [cont offset]
JUMPI

after optimization
DUP1
PUSH1 [revert offset]
JUMPI
```

Disclaimer: There have been several bugs with security implications related to optimizations. For this reason, Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised. High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten -generated solc-js compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. Please measure the gas savings from optimizations, and carefully weigh them against the possibility of an optimization-related bug. Also, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

[G-12] Constructors can be marked payable

Payable functions cost less gas to execute, since the compiler does not have to add extra checks to ensure that a payment wasn't provided. A constructor can safely be marked as payable, since only the deployer would be able to pass funds, and the project itself would not pass any funds.

Here are some of the instances entailed:

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/aave/ATokenFiatCollateral.sol#L42

```
    constructor(CollateralConfig memory config, uint192 revenueH
```

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/EURFiatCollateral.sol#L23-L27

```
    constructor(
        CollateralConfig memory config,
        AggregatorV3Interface targetUnitChainlinkFeed_,
        uint48 targetUnitOracleTimeout_
    ) FiatCollateral(config) {
```

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/FiatCollateral.sol#L62-L70

```
        constructor(CollateralConfig memory config)
            Asset(
                config.priceTimeout,
                config.chainlinkFeed,
                config.oracleError,
                config.erc20,
                config.maxTradeVolume,
                config.oracleTimeout
            )
```

[G-13] Use assembly for small keccak256 hashes, in order to save gas

If the arguments to the encode call can fit into the scratch space (two words or fewer), then it's more efficient to use assembly to generate the hash (80 gas):
keccak256(abi.encodePacked(x, y)) -> assembly {mstore(0x00, a); mstore(0x20, b); let hash := keccak256(0x00, 0x40); }

Here are some of the instances entailed:

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/aave/StaticATokenLM.sol#L285-L286

```
                        keccak256(bytes(name())),
                        keccak256(EIP712_REVISION),
```

[G-14] Reduce gas usage by moving to Solidity 0.8.19 or later

Please visit the following link for substantiated details:

https://soliditylang.org/blog/2023/02/22/solidity-0.8.19-release-announcement/#preventing-dead-code-in-runtime-bytecode

And, here are some of the instances entailed:

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/pl

ugins/assets/aave/StaticATokenLM.sol#L2

```
pragma solidity 0.6.12;
```

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/aave/StaticATokenErrors.sol

```
pragma solidity 0.6.12;
```

[G-15] Using this to access functions results in an external call, wasting gas

External calls have an overhead of 100 gas, which can be avoided by not referencing the function using this. Contracts are allowed to override their parents' functions and change the visibility from external to public, so make this change if it's required in order to call the function internally.

Here are some of the instances entailed:

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/AppreciatingFiatCollateral.sol#L104

```
        try this.tryPrice() returns (uint192 low, uint192 high,
```

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/plugins/assets/Asset.sol

```
89:        try this.tryPrice() returns (uint192 low, uint192 hig

113:        try this.tryPrice() returns (uint192 low, uint192 hi

129:        try this.tryPrice() returns (uint192 low, uint192 hi
```

Audit Analysis

For this audit, 2 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The report highlighted below by **RaymondFam** received the top score from the judge.

The following warden also submitted a report: 0xA5DF.

Introduction

The following is a detailed analysis of the Reserve Protocol, with a comprehensive exploration of the codebase, architecture, quality, centralization risks, mechanisms, and potential systemic risks. In this report, six key vulnerabilities have been identified and ranked by severity, with three identified as high, and the remaining three as medium. The potential issues range from token loss during transfers to early exploits, reentrancy vulnerabilities, incorrect collateral pricing, and lack of protection during withdrawals. Each vulnerability is thoroughly explained with corresponding recommendations to mitigate the associated risks. The report further covers the evaluation approach, providing insightful suggestions for improving the codebase and minimizing centralization risks. The mechanism review and an assessment of systemic risks offer an understanding of the potential threats that need to be managed for long-term system stability.

Comments for the judge to contextualize my findings

A total of 3 highs, 3 mediums along with additional inputs in the Gas and QA reports have been submitted. It's worth noting that loopholes still abound depending on how you would visualize the codebase both short and long terms. Here is the breakdown of the HM findings in their condensed forms:

1. Potential Loss of Rewards During Token Transfers in StaticATokenLM.sol (High)
   The report identifies a potential issue in the Reserve Protocol, where the process of transferring tokens could lead to permanent loss of rewards for the sender due to the order of function calls and behavior of certain functions. The issue arises primarily from the `_getPendingRewards` function not accounting for all recent rewards from the `INCENTIVES_CONTROLLER` when the `fresh` parameter is set to `false`. This miscalculation could then negatively affect the output of the

`_updateUserSnapshotRewardsPerToken` function, resulting in some rewards being missed, particularly affecting the `Backing Manager` during `RToken` redemptions. The issue could also affect token transfers to successful auction bidders. The report recommends mitigating this by introducing an additional call to update the state of rewards before any token transfer occurs, specifically within the `_beforeTokenTransfer` function, to ensure accurate internal accounting of rewards and prevent loss or premature gain during transfers.

2. Potential Early Exploit in Morho-Aave ERC4626 Implementation (High) The report describes an exploit in a blockchain smart contract where an attacker could potentially gain unauthorized control over a significant portion of the assets stored within the contract's "vault". This is possible when the vault is initially empty, and an attacker deposits a negligible amount before a legitimate user's deposit, thereby owning shares while the total vault asset is still low. The attacker then donates a large amount to the vault, causing a precision error which leads to other users receiving no shares despite depositing assets. The attacker redeems their shares, possibly gaining a significant portion of the vault's assets. This vulnerability arises from the flawed inheritance structure of the underlying smart contract and its functions, particularly the `MorphoTokenisedDeposit._decimalsOffset()` function. A proof-of-concept is given which shows how an attacker can implement this exploit in five steps. The report recommends mitigating this risk by hardcoding `MorphoTokenisedDeposit._decimalsOffset()` to return `9` instead of `0`.

3. Cross-Function Reentrancy Vulnerability Leading to Unintended Token Minting in `RewardableERC20Wrapper.deposit` (High) The report outlines a substantial reentrancy vulnerability within the `RewardableERC20Wrapper` contract, particularly the `deposit()` function. This vulnerability, which can be exploited through direct and cross-function reentrancy attacks, becomes evident when ERC777 tokens or other token types with "hook" features serve as the underlying token. Successful exploitation could trigger unwarranted token minting, leading to a skewed token supply, thereby compromising the contract's integrity and impacting individual token value. This susceptibility is due to the `_mint(_to, _amount)` operation execution prior to the `underlying.safeTransferFrom(msg.sender, address(this), _amount)` call within the `deposit()` function. Such a sequence allows an ERC777 token to initiate a reentrant call back to `deposit()`, thereby allowing token minting before the contract's state is adequately updated. To mitigate this, the report suggests implementing a reentrancy guard for all public and external functions

that modify the contract's state and call external contracts. It also recommends refactoring the `deposit()` function to adhere to the check-effects-interactions pattern, conducting state changes post external calls.

4. Risk of Incorrect Collateral Pricing in Case of Aggregator Reaching minAnswer (Medium) The report highlights a significant issue within Chainlink aggregators related to the built-in circuit breaker, which can cause the oracle to continuously return the `minPrice` rather than the actual asset price during substantial price drops. The problem occurs when an asset's price goes below its `minPrice` ; the protocol then overvalues the token at the `minPrice` , leading to overvalued function calls. The issue becomes prominent when the `defaultThreshold` is set in a way that keeps the `pegPrice` between `pegBottom` and `pegTop` , which can potentially lead to an excessively large issuance of `RTokens` , creating an unnoticed unhealthy collateral basket. Although a combination of oracles, such as Chainlink and Band, could potentially prevent this situation, a malicious user could still exploit Band by DDOSing relayers to block price updates. To address this issue, the report suggests cross-checking the returned price in the `OracleLib.price` against `minPrice/maxPrice` , and triggering a revert if the price falls outside these boundaries, thus ensuring an accurate price representation.
5. Lack of protection when withdrawing Static Atoken (Medium) The report reveals a significant issue with the Aave plugin regarding its ever-increasing exchange rate. It shows that while an early wrap of AToken results in more Static Atoken minting, withdrawal timing can greatly impact the amount of AToken redeemed due to the nonlinear growth of the exchange rate. This could lead to unequal percentage gains for users who perform similar calls around the same time. Furthermore, rewrapping the AToken may not be beneficial if it encounters a sudden surge, consequently thwarting the initial objective. The highlighted code snippets show that choosing `staticAmount > 0` could result in a lesser amount of AToken being withdrawn when the `currentRate` is stagnant. In contrast, choosing `dynamicAmount > 0` could lead to burning a higher than expected amount of Static Atoken. To mitigate this, the report suggests implementing slippage protection on `StaticATokenLM._withdraw` so that users could set the minimum AToken amount to receive or the maximum Static Atoken amount to burn.
6. Lack of protection when caling `CusdcV3Wrapper._withdraw` (Medium) The report discusses potential risks associated with the `wComet` unwrapping process in the `CusdcV3Wrapper` contract. It notes that users who invoke

`CusdcV3Wrapper._withdraw` simultaneously could experience different percentage gains due to the non-linear rebasing of `comet`. Moreover, the report suggests that the rate-determining `getUpdatedSupplyIndicies()` function is inaccessible to most users who make partial withdrawals, obscuring the ideal unwrapping moment. The document provides a detailed code walkthrough, demonstrating how the value of `baseSupplyIndex` influences the burn amount, with lower values resulting in higher `principalValueSupply` and therefore a smaller `burn` quantity. To remedy these issues, the report proposes implementing slippage protection on `CusdcV3Wrapper._withdraw`, allowing users to set the minimum `comet` to receive or the maximum `wComet` to burn.

Approach taken in evaluating the codebase

Going through the recommended specs and links is of paramount importance prior to going over the codebases line upon line. I started with the generic contracts and then moved on to tackling the specific plugins one after another. It has also helped me smell out some bugs by reading past related audit reports as I pieced together the puzzles. Deep diving into the code logic was fun and satisfying when noticing how the flaws could be so exploited.

Architecture recommendations

The codebase could do better with adequate and complete NatSpec along with some thorough flow charts to help link up essential flows/calls. There have been some redundant and unneeded codes presented in the Gas report. Touching up the codebase with the low and non-critical feedback from the QA report would also help amplify code robustness.

Codebase quality analysis

The Reserve Protocol features one of the most sophisticated and well-structured codebases in the industry. Having prior knowledge of the previous audits will not make it enough to revisit the documentation and specifications. You would assuredly gain a better and often another facet of understanding of the business logic particularly when it relates to Dimensional Analysis.

Centralization risks

There are some centralized risks associated with the vendors but they are deemed out of scope here. Nonetheless, these vendors are mostly reputable third-party contracts that have relatively been battled tested and should not pose much of a concern at this juncture. Perhaps, the protocol should look into implementing some form of emergency contract and/or key function pausing/freezing and be prepared for the worst.

Mechanism review

The Reserve Protocol, with its intricate design and interlocking mechanisms, presents unique challenges for ensuring the stability and security of its network. The protocol has addressed several key concerns raised in previous reports, showcasing its commitment to maintaining a safe and reliable platform for its users.

However, the protocol’s complexity necessitates ongoing vigilance. The Token Transfer Mechanism, Vault Control Mechanism, and Withdrawal Mechanisms, while remedied, need constant monitoring to prevent the re-emergence of issues such as inaccurate accounting of rewards or unauthorized asset control.

Moreover, the Token Minting and Price Retrieval Mechanisms require careful handling to avoid potential reentrancy attacks and misvaluation of function calls. Any discrepancies in these systems could lead to an unpredictable token supply and excessive issuance of RTokens, thus destabilizing the entire ecosystem.

Systemic risks

Limited risks will always prevail considering the majority of current designs have not gone live yet. The previous audit on governance currently falling also under the plugins/*, when more elaborately worked upon and implemented, should help defray all anticipated threats. I think the adoption of Revenue Hiding via AppreciatingFiatCollateral.sol is a smart approach overall to minimize the likelihood of having collaterals defaulted to DISABLED. Coupled with good collateral rewards revenue incentives, the high participation of RSR stakers that enhances over-collateralization would assuredly make the system fully established and intact from getting anywhere near to an undesirable haircut.

Time spent

72 hours

Mitigation Review

Introduction

Following the C4 audit, 3 wardens (ronnyx2017, bin2chen and RaymondFam) reviewed the mitigations for all identified issues. Additional details can be found within the Reserve Mitigation Review repository.

Overview of Changes

Summary from the Sponsor:

Units and price calculations in LSD collateral types were fixed. `CurveVolatileCollateral` was removed entirely. Decimals fixed in wrapped `cUSDCv3`. RToken Asset pricing issues fixed, (`0, FIX_MAX`) enforced as "unpriced". Reward remainder held until next claim instead of lost.

Mitigation Review Scope

Branch

https://github.com/reserve-protocol/protocol/tree/master

Individual PRs

URL	Mitigation of	Purpose
https://github.com/reserve-protocol/protocol/pull/899	H-01	Fixes units and price calculations in cbETH, rETH, ankrETH collateral plugins.
https://github.com/reserve-protocol/protocol/pull/896	H-02	Removes `CurveVolatileCollateral`.
https://github.com/reserve-protocol/protocol/pull/930	H-03	Skip reward claim in `_checkpoint` if shutdown.
https://github.com/reserve-protocol/protocol/pull/896	M-01	Removes `CurveVolatileCollateral`.
https://github.com/reserve-protocol/protocol/pull/889	M-02	Use decimals from underlying Comet.
https://github.com/reserve-protocol/protocol/pull/916	M-03	Acknowledged and documented.

URL	Mitigati on of	Purpose
https://github.com/reserve-protocol/protocol/pull/896	M-04	Roll over remainder to next call.
https://github.com/reserve-protocol/protocol/pull/896	M-05	Add call to `emergencyWithdraw`.
https://github.com/reserve-protocol/protocol/pull/917	M-06	Enforce (`0, FIX_MAX`) as “unpriced” during oracle timeout.
https://github.com/reserve-protocol/protocol/pull/917	M-08	Unpriced on oracle timeout.
https://github.com/reserve-protocol/protocol/pull/917	M-09	Enforce (`0, FIX_MAX`) as “unpriced” during oracle timeout.

Out of Scope

URL	Mitigation of	Purpose
	M-07	Acknowledged. See details in comment.
https://github.com/reserve-protocol/protocol/pull/896	M-10	Acknowledged, documented.
https://github.com/reserve-protocol/protocol/pull/920	M-11	Acknowledged. Details in comment.
https://github.com/reserve-protocol/protocol/pull/920	M-12	Acknowledged. Details in comment.
	M-13	Acknowledged. Details in comment.
	M-14	Acknowledged. Details in comment.
	M-15	Acknowledged. Details in comment.

Mitigation Review Summary

Original Issue	Status	Full Details
H-01	🟢 Mitigation Confirmed	Reports from ronnyx2017, bin2chen and RaymondFam

Original Issue	Status	Full Details
H-02	🟢 Mitigation Confirmed	Reports from ronnyx2017, RaymondFam and bin2chen
H-03	🟢 Mitigation Confirmed	Reports from ronnyx2017, bin2chen and RaymondFam
M-01	🟢 Mitigation Confirmed	Reports from RaymondFam, ronnyx2017 and bin2chen
M-02	🟢 Mitigation Confirmed	Reports from RaymondFam, ronnyx2017 and bin2chen
M-03	🟢 Mitigation Confirmed	Reports from bin2chen, RaymondFam and ronnyx2017
M-04	🟢 Mitigation Confirmed	Reports from ronnyx2017, RaymondFam and bin2chen
M-05	🟢 Mitigation Confirmed	Reports from ronnyx2017, bin2chen and RaymondFam
M-06	🟢 Mitigation Confirmed	Reports from ronnyx2017, RaymondFam and bin2chen
M-08	🟢 Mitigation Confirmed	Reports from ronnyx2017, RaymondFam and bin2chen
M-09	🟢 Mitigation Confirmed	Reports from ronnyx2017, RaymondFam and bin2chen

There were also 3 new Medium severity issues surfaced by the wardens. See below for details regarding the new issues.

`getReward()` is not called after shutdown, which could lead to incorrect reward accumulation

Submitted by bin2chen

Severity: Medium

In the previous implementation, after shutdown, checkpoints are stopped `reward.reward_integral_for[user]`. No updates resulted in new users getting more rewards and possible theft of rewards.

Mitigation

PR 930

Modify that `checkpoints` are already executed, not just calling `IRewardStaking(convexPool).getReward(address(this), true);` the mitigation resolved the original issue.

Suggestion

By not calling `convexPool.getReward()`, there is a slight loss of rewards for transferred users. The feeling is that there is no need to ignore this call, `convexPool.getReward()`, just don't revert if shutdown.

cccz (judge) commented:

> `getReward()` is not called after shutdown, which could lead to incorrect reward accumulation. Consider the simple scenario where Alice is the only depositor.

1. Alice deposits 1000 tokens.
2. 100 reward tokens are generated, Alice claims the reward, `convexPool.getReward()` is called, and Alice receives 100 reward tokens.
3. Another 100 reward tokens are generated, and the owner shuts down the Wrapper.
4. Alice executes `withdraw(1000)`. Since `convexPool.getReward()` is not triggered, their accumulated rewards will not be increased, but the balance will be changed to 0. So `another 100 reward tokens are generated` will be lost.

Rewards can be incorrectly distributed due to rounding rollover

Submitted by RaymondFam*, also found by* bin2chen

Severity: Medium

Lines of code

https://github.com/reserve-protocol/protocol/blob/9ee60f142f9f5c1fe8bc50eef915cf33124a534f/contracts/pl

ugins/assets/erc20/RewardableERC20.sol#L86

Impact

The previously identified vulnerability of potential rounding issues during reward calculations has not been fully mitigated. The current strategy to keep remainders and use them in subsequent `_claimAndSyncRewards()` calls does not adequately address the issue when the `rewardToken` has a decimal smaller than 6 and/or the total reward tokens entailed is much smaller. This could lead to significant truncation losses as the remainder rolls over until it's large enough to overcome truncation and unfairly disadvantaging users; particularly those exiting the investment earlier, as they would miss out on a sizable amount of reward. This rounding issue, if left unresolved, can erode trust and potentially open up the system to arbitrage opportunities, further exacerbating the loss of rewards for regular users.

Proof of Concept

Scenario:

Let's assume `rewardToken` still has 6 decimals but there are only 0.5 million `rewardToken` to be distributed for the year and `_claimAndSyncRewards()` is called every minute. And, `totalSupply = 10^6` with 18 decimals.

The expected rewards for 1 min are `500000 / 365 / 24 / 60 = 0.95 rewardToken = 950000 wei`.

Initially, assume `balanceAfterClaimingRewards = 1950000` (wei), and `_previousBalance = 1000000` (wei), making `delta = 950000` (wei).

`deltaPerShare` will be calculated as: `(950000 * 10^18) / (10^6 * 10^18) = 0`

Now, `balanceAfterClaimingRewards` is updated to: `previous balance + (deltaPerShare * totalSupply / one) = 1000000 + (0 * (10^6 * 10^18) / 10^18) = 1000000 + 0 = 1000000 (wei)`

As illustrated, the truncation issue causes `deltaPerShare` to equal `0`. This will lead to a scenario where the rewards aren't distributed accurately among users;

particularly affecting those who exit earlier before the remainder becomes large enough to surpass truncation.

In a high-frequency scenario where `_claimAndSyncRewards` is invoked often, users could miss out on a significant portion of rewards, showcasing the inadequacy of the proposed mitigation in handling the rounding loss effectively.

Mitigation

Using a bigger multiplier as the original report suggested seems viable, but finding a suitably discrete factor could be tricky.

While keeping the current change per PR #896, I suggest adding another step by normalizing both `delta` and `_totalSupply` to `PRICE_DECIMALS`, i.e. 18, which will greatly minimize the prolonged remainder rollover. The intended decimals may be obtained by undoing the normalization when needed. Here are the two useful functions (assuming `decimals` is between 1 to 18) that could help handle the issue, but it will require further code refactoring on `_claimAndSyncRewards()` and `_syncAccount()`.

```
    /// @dev    Convert decimals of the value to price decimals
    function _toPriceDecimals(uint128 _value, uint8 decimals, ad
        internal
        view
        returns (uint256 value)
    {
        if (PRICE_DECIMALS == decimals) return uint256(_value);
        value = uint256(_value) * 10 ** (PRICE_DECIMALS - decima
    }

    /// @dev    Convert decimals of the value from the price dec
    function _fromPriceDecimals(uint256 _value, uint8 decimals, 
        internal
        view
        returns (uint128 value)
    {
        if (PRICE_DECIMALS == decimals) return _toUint128(_value
        value = _toUint128(_value / 10 ** (PRICE_DECIMALS - deci
    }
```

Assessed type

Decimal

ronnyx2017 (warden) commented:

```
deltaPerShare will be calculated as: (950000 * 10^18) / (10^6 * 
```

> It's based on `uint256 deltaPerShare = (delta * one) / _totalSupply;` and `uint256 delta = balanceAfterClaimingRewards - _previousBalance;`.

> But the `delta` is not always `950000`. It will accumulate over time because the `lastRewardBalance = balanceAfterClaimingRewards = _previousBalance + (deltaPerShare * _totalSupply) / one`.

> So the max loss will be less than `10^6` wei - is my understanding, correct? Have I missed anything?

bin3chen (warden) commented:

> My personal understanding is that this is an acceptable simple solution.

> See mitigation description for details in Issue `#9`:

```
This is a simple way to fix. Some rewards might be locked inside
A more reasonable approach would be to increase the precision, e
```

RaymondFam (warden) commented:

> The rollover issue could be quite pronounced/prolonged if all of the factors below were to kick in together:

> A much smaller numerator than anticipated due to:

1. e.g. 10k `rewardToken` to be distributed for the year
2. A much smaller decimal (1 - 5) associated with `rewardToken`

> A much bigger denominator than anticipated due to a bigger `totalSupply` entailed say in tens of millions or even larger.

cccz (judge) commented:

> According to the Mitigation Review Guidelines, I'm inclined to consider this a new issue.

> The original issue causes rewards to be locked in contract, and the mitigation solves it, but introduces the new issue of rewards being incorrectly distributed.

> The potential loss caused by this issue is related to the supply and price of deposit token and the decimals and price of the reward token.

> In the extreme case where the deposit token is SHIB and the reward token is WBTC (8 decimals), the loss may be unacceptable (1 WBTC reward for per 700 USD SHIB loss).

User token locked due to `StargateRewardableWrapper` no longer being able to execute `StargateRewardableWrapper.withdraw()`

Submitted by bin2chen

In the previous implementation, when `stakingContract.totalAllocPoint = 0` `stakingContract.withdraw()` and `stakingContract.deposit()` will div 0 , `revert` . This results in `StargateRewardableWrapper` no longer being able to execute `StargateRewardableWrapper.withdraw()` , as the user's token is locked.

Mitigation

PR 896

Determine if `poolInfo.allocPoint` is equal to `0` . If equal to `0` , use `stakingContract.emergencyWithdraw()` instead of `stakingContract.deposit()` to avoid reverting. The mitigation resolved the original issue.

Suggestion

Since `allocPoint==0` is used instead of `totalAllocPoint==0`, there may be a case where `allocPoint == 0` but `totalAllocPoint> 0`. However, the modified version still uses `stakingContract.emergencyWithdraw()`, which discards all rewards. It is recommended that if `totalAllocPoint> 0`, we can execute the `stakingContract.deposit(0)` to retrieve the reward first, then execute `stakingContract.emergencyWithdraw()`.

bin2chen (warden) commented:

> This problem is due to use `allocPoint==0` is used instead of `totalAllocPoint==0`. Some of the rewards are lost in this scenario.

> Assumption:

> The current: `poolInfo.allocPoint = 50`, `totalAllocPoint = 100` (have another pool).

1. After a certain period of time, `poolInfo.pendingRewards = 100`.
2. `stakingContract.set()` changes `poolInfo.allocPoint = 0`, but `totalAllocPoint` is still `50`. (`set()` will trigger accumulation pending rewards first).
3. `StargateRewardableWrapper.claim()` -> `stakingContract.emergencyWithdraw()` will discard any pending rewards.

> In step 3, if `totalAllocPoint > 0`, we can use `stakingContract.deposit(0)` to retrieve the pending rewards first, then call `stakingContract.emergencyWithdraw()` to avoid discarding the `pendingRewards`.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and

Top