Introducing Code4rena Blue: Dedicated defense. Competitive bounties. Independent judging.
Learn more →

Reserve Protocol - Invitational Findings & Analysis Report

2023-11-13

Table of contents

- Overview
  - About C4
  - Wardens
- Summary
- Scope
- Severity Criteria
- High Risk Findings (2)
  - [H-01] Custom redemption might revert if old assets were unregistered
  - [H-02] A new era might be triggered despite a significant value being held in the previous era
- Medium Risk Findings (12)
  - [M-01] A Dutch trade could end up with an unintended lower closing price
  - [M-02] The broker should not be fully disabled by GnosisTrade.reportViolation
  - [M-03] In case `Distributor.setDistribution` use, revenue from rToken RevenueTrader and rsr token RevenueTrader should be distributed
  - [M-04] FurnaceP1.setRatio will work incorrectly after call when frozen

- [M-05] Lack of claimRewards when manageToken in RevenueTrader
- [M-06] Oracle timeout at rebalance will result in a sell-off of all RSRs at 0 price
- [M-07] Sell reward `rTokens` at low price because of skiping `furnace.melt`
- [M-08] Stake before unfreeze can take away most of rsr rewards in the freeze period
- [M-09] `cancelUnstake` lack `payoutRewards` before mint shares
- [M-10] An oracle deprecation might lead the protocol to sell assets for a low price
- [M-11] Attacker can disable basket during un-registration, which can cause an unnecessary trade in some cases
- [M-12] Custom redemption can be used to get more than RToken value, when an upwards depeg occurs

- Low Risk and Non-Critical Issues
  - 01 A redeemer might get 'front-run' by a freezer
  - 02 Leaky refresh math is incorrect
  - 03 Reorg attack
  - 04 `distributeTokenToBuy()` can be called while paused/frozen
  - 05 `redeemCustom` allows the use of the zero basket
  - 06 `refreshBasket` can be called before the first prime basket was set
  - 07 `MIN_AUCTION_LENGTH` seems too low
  - 08 If a token that yields RSR would be used as collateral then 100% of the yield would go to StRSR
  - 09 Protocol might not be able to compromise basket when needed
- Gas Optimizations
  - G-01 At `toColl()` and `toAsset()` use the mapping to check if asset exists
  - G-02 Get `targetIndex` from mapping instead of iterating

    - G-03 Use `furnace` instead of `main.furnace()`
    - G-04 Deployer.implementations can be immutable
    - G-05 Update `lastWithdrawRefresh` only if it has changed
    - G-06 Require array to be sorted and use `sortedAndAllUnique` at `BackingManager.forwardRevenue()`
    - G-07 Caching storage variable and function calls
    - G-08 `BasketLib.nextBasket()` refactoring
    - G-09 `BasketLib.nextBasket()` caching
    - G-10 `sellAmount` at `DutchTrade.settle()`
    - G-11 `quoteCustomRedemption()` loop
    - G-12 Use custom errors instead of string errors
- Mitigation Review
    - Introduction
    - Mitigation Review Scope
    - Mitigation Review Summary
    - M-02 Mitigation Error: `dutchTradeDisabled[erc20]` gives governance an incentive to disable RSR auctions
    - M-02 Mitigation Error: Attacker might disable trading by faking a report violation
    - M-03 Mitigation Error: Funds aren’t distributed before changing distribution
    - M-04 Mitigation Error: Furnace would melt less than intended
- Disclosures

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 June 15—June 29 2023.

Following the C4 audit, 3 wardens (0xA5DF, ronnyx2017, and rvierdiiev) 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, 6 wardens contributed reports:

1. 0xA5DF
2. ronnyx2017
3. rvierdiiev
4. RaymondFam
5. carlitox477
6. hihen

This Audit was judged by 0xean.

Final report assembled by PaperParachute and liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 14 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 12 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 4 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 repository, and is composed of 12 smart contracts written in the Solidity programming language and includes 2126 lines of Solidity code.

In addition to the known issues identified by the project team, the C4udit tool was ran prior to the audit launch. This generated the Automated Findings report and all findings therein were classified as out of scope.

Note: the automated findings report also included one medium-severity finding *that has been acknowledged by the sponsor and confirmed by the judge.*

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 (2)

[H-01] Custom redemption might revert if old assets were unregistered

Submitted by 0xA5DF

`quoteCustomRedemption()` works under the assumption that the maximum size of the `erc20sAll` should be `assetRegistry.size()`, however there can be cases where an asset was unregistered but still exists in an old basket, making the size of the old basket greater than `assetRegistry.size()`. In that case the function will revert with an index out of bounds error.

Impact

Users might not be able to use `redeemCustom` when needed.

I think this should be considered high severity, since being able to redeem the token at all time is an essential feature for the protocol that's allowed also while frozen. Not being able to redeem can result in a depeg or in governance becoming malicious and stealing RToken collateral.

Proof of Concept

Consider the following scenario:

- RToken deployed with 0.9 USDC, 0.05 USDT, 0.05 DAI
- Governance passed a vote to change it to 0.9 DAI and 0.1 USDC and un-register USDT
- Trading is paused before execution, so the basket switch occurs but the re-balance can't be executed. Meaning the actual assets that the backing manager holds are in accordance with the old basket
- A user wants to redeem using the old basket, but custom redemption reverts

As for the revert:

- `erc20sAll` is created here with the length of `assetRegistry.size()`, which is 2 in our case.
- Then in this loop the function tries to push 3 assets into `erc20sAll` which will result in an index-out-of-bonds error

(the function doesn't include in the final results assets that aren't registered, but it does push them too into `erc20sAll`)

Recommended Mitigation Steps

Allow the user to specify the length of the array `erc20sAll` to avoid this revert

[0xean (judge) commented]():

> I believe this to be a stretch for high severity. It has several pre-conditions to end up in the proposed state and I do believe it would be entirely possible for governance to change back to the original state (USDC, USDT, DAI), so assets wouldn't be lost and the impact would more be along the lines of a temporary denial of service.

> Look forward to warden and sponsor comments.

[tbrent (Reserve) confirmed and commented]():

> @0xA5DF - nice find! Thoughts on an alternative mitigation?
>
> - Could move L438 to just after L417, so that `erc20sAll` never includes unregistered ERC20s
> - Would probably have to cache the assets as `assetsAll` for re-use around L438
> - Has side-effect of making the ERC20 return list never include unregistered ERC20s. Current implementation can return a 0 value for an unregistered ERC20. This is properly handled by the RToken contract, but still, nice-to-have.

[0xA5DF (warden) commented]():

> Hey @tbrent -
> That can work as well, the only downside I can think of is that in case there's an asset that's not registered and is repeated across different baskets - the `toAsset()` would be called multiple times for that asset (while under the current implementation and under the mitigation I've suggested it'll be called only once), this would cost about 300 gas units per additional call (100 for the call, 2 `sload`s to a warm slot inside the call itself)

[tbrent (Reserve) commented]():

> @0xA5DF - Noted, good point.

[tbrent (Reserve) confirmed]()

Oxean (judge) commented:

> @tbrent - do you care to comment on your thoughts on severity? I am leaning towards M on this, but it sounds like you believe it is correct as labeled (high).

tbrent (Reserve) commented:

> @Oxean - Correct, I think high is appropriate.

Reserve mitigated:

> Fix `redeemCustom`.
> PR: https://github.com/reserve-protocol/protocol/pull/857

Status: Mitigation confirmed. Full details in reports from ronnyx2017, 0xA5DF, and rvierdiiev - and also shared below in the Mitigation Review section.

[H-02] A new era might be triggered despite a significant value being held in the previous era

Submitted by 0xA5DF

https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/StRSR.sol#L441-L444
https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/StRSR.sol#L457-L460

When RSR seizure occurs the staking and drafting rate is adjusted accordingly, if any of those rates is above some threshold then a new era begins (draft or staking era accordingly), wiping out all of the holdings of the current era. The assumption is that if the rate is above the threshold then there's not much staking or drafts left after the seizure (and therefore it makes sense to begin a new era). However, there might be a case where a previous seizure has increased the staking/draft rate close to the threshold, and then even a small seizure would make it cross this threshold. In that case the total value of staking or drafts can be very high, and they will all be wiped out by starting a new era.

Impact

Stakers will lose their holdings or pending drafts.

Proof of Concept

Consider the following scenario:

- Max stake rate is 1e9
- A seizure occurs and the new rate is now 91e7
- Not much staking is left after the seizure, but as time passes users keep staking bring back the total stakes to a significant value
- A 10% seizure occurs, this causes the staking rate to cross the threshold (getting to 1.01e9) and start a new era

This means the stakings were wiped out despite holding a significant amount of value, causing a loss for the holders.

Recommended Mitigation Steps

This one is a bit difficult to mitigate. One way I can think of is to add a 'migration' feature, where in such cases a new era would be created but users would be able to transfer the funds that they held in the previous era into the new era. But this would require some significant code changes and checking that this doesn't break anything or introduces new bugs.

tbrent (Reserve) commented:

> @0xA5DF thoughts on a governance function that requires the ratio be out of bounds, that does `beginEra()` and/or `beginDraftEra()` ?

> The idea is that stakers can mostly withdraw, and since governance thresholds are all percentage, vote to immolate themselves and re-start the staking pool. I think it should treat `beginEra()` and `beginDraftEra()` separately, but I'm not confident in that yet.

tbrent (Reserve) acknowledged and commented:

> We're still not sure how to mitigate this one. Agree it should be considered HIGH and a new issue.

Reserve mitigated:

> Adds governance function to manually push the era forward.
> PR: https://github.com/reserve-protocol/protocol/pull/888

Status: Mitigation confirmed. Full details in reports from 0xA5DF, ronnyx2017, and rvierdiiev - and also shared below in the Mitigation Review section.

Medium Risk Findings (12)

[M-01] A Dutch trade could end up with an unintended lower closing price

Submitted by RaymondFam

https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/plugins/trading/DutchTrade.sol#L160
https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/RevenueTrader.sol#L46
https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/BackingManager.sol#L81

`notTradingPausedOrFrozen` that is turned on and off during an open Dutch trade could have the auction closed with a lower price depending on the timimg, leading to lesser capability to boost the Rtoken and/or stRSR exchange rates as well as a weakened recollaterization.

Proof of Concept

Here's the scenario:

1. A 30 minute Dutch trade is opened by the Revenue trader selling a suplus token for Rtoken.
2. Shortly after the price begins to decrease linearly, Alice calls `bid()` . As can be seen in line 160 of the code block below, `settleTrade()` is externally called on the `origin` , RevenueTrader.sol in this case:

```
    function bid() external returns (uint256 amountIn) {
        require(bidder == address(0), "bid already received");

        // {qBuyTok}
        amountIn = bidAmount(uint48(block.timestamp)); // enforc

        // Transfer in buy tokens
        bidder = msg.sender;
        buy.safeTransferFrom(bidder, address(this), amountIn);

        // status must begin OPEN
        assert(status == TradeStatus.OPEN);

        // settle() via callback
160:        origin.settleTrade(sell);

        // confirm callback succeeded
        assert(status == TradeStatus.CLOSED);
    }
```

3. However, her call is preceded by `pauseTrading()` invoked by a `PAUSER` , and denied on line 46 of the function below:

https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/RevenueTrader.sol#L43-L52

```
    function settleTrade(IERC20 sell)
        public
        override(ITrading, TradingP1)
46:        notTradingPausedOrFrozen
        returns (ITrade trade)
    {
        trade = super.settleTrade(sell); // nonReentrant
        distributeTokenToBuy();
```

```
            // unlike BackingManager, do _not_ chain trades; b2b tra
        }
```

4. As the auction is nearing to `endTime`, the `PAUSER` calls [`unpauseIssuance()`].
5. Bob, the late comer, upon seeing this, proceeds to calling `bid()` and gets the sell token for a price much lower than he would initially expect before the trading pause.

Recommended Mitigation Steps

Consider removing `notTradingPausedOrFrozen` from the function visibility of `RevenueTrader.settleTrade` and `BackingManager.settleTrade`. This will also have a good side effect of allowing the settling of a Gnosis trade if need be. Collectively, the settled trades could at least proceed to helping boost the RToken and/or stRSR exchange rates that is conducive to the token holders redeeming and withdrawing. The same shall apply to enhancing recollaterization, albeit future tradings will be halted if the trading pause is still enabled.

[0xean (judge) commented]:

> This also seems like QA. It outlines a very specific set of events that are very unlikely to occur during production scenarios and would additionally come down to admin misconfiguration / mismanagement. will wait for sponsor comment, but most likely downgrade to QA.

- The PAUSER role should be assigned to an address that is able to act quickly in response to off-chain events, such as a Chainlink feed failing. It is acceptable for there to be false positives, since redemption remains enabled.

> It is good to consider this quote from the documentation stating that pausing may have false positives.

[tbrent (Reserve) confirmed and commented]:

> @0xean - We believe a malicious pauser attack vector is dangerous enough that the issue is Medium and deserves a mitigation. Agree with suggested mitigation.

[Reserve mitigated]:

> Allow settle trade when paused or frozen.
> PR: https://github.com/reserve-protocol/protocol/pull/876

Status: Mitigation confirmed. Full details in reports from rvierdiiev, 0xA5DF, and ronnyx2017 - and also shared below in the Mitigation Review section.

[M-02] The broker should not be fully disabled by GnosisTrade.reportViolation

Submitted by RaymondFam

https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/plugins/trading/GnosisTrade.sol#L202
https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/Broker.sol#L119-L123

GnosisTrade and DutchTrade are two separate auction systems where the failing of either system should not affect the other one. The current design will have `Broker.sol` disabled when `reportViolation` is invoked by `GnosisTrade.settle()` if the auction's clearing price was below what we assert it should be.

https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/plugins/trading/GnosisTrade.sol#L202

```
                broker.reportViolation();
```

https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/Broker.sol#L119-L123

```
    function reportViolation() external notTradingPausedOrFrozen
        require(trades[_msgSender()], "unrecognized trade contra
        emit DisabledSet(disabled, true);
        disabled = true;
    }
```

Consequently, both `BackingManager` and `RevenueTrader (rsrTrader and rTokenTrader)` will not be able to call `openTrade()`:

https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/Broker.sol#L97-L98

```
        function openTrade(TradeKind kind, TradeRequest memory req)
            require(!disabled, "broker disabled");
            ...
```

till it's resolved by the governance:

https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/Broker.sol#L180-L183

```
        function setDisabled(bool disabled_) external governance {
            emit DisabledSet(disabled, disabled_);
            disabled = disabled_;
        }
```

Proof of Concept

The following `Trading.trytrade()` as inherited by `BackingManager` and `RevenueTrader` will be denied on line 121, deterring recollaterization and boosting of Rtoken and stRSR exchange rate. The former deterrence will have `Rtoken.redeemTo` and `StRSR.withdraw` (both **requiring** `fullyCollateralized`) denied whereas the latter will have the Rtoken and stRSR holders divested of intended gains.

https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/mixins/Trading.sol#L113-L126

```
    function tryTrade(TradeKind kind, TradeRequest memory req) i
        /*  */
        IERC20 sell = req.sell.erc20();
        assert(address(trades[sell]) == address(0));

        IERC20Upgradeable(address(sell)).safeApprove(address(bro
        IERC20Upgradeable(address(sell)).safeApprove(address(bro

121:        trade = broker.openTrade(kind, req);
        trades[sell] = trade;
        tradesOpen++;

        emit TradeStarted(trade, sell, req.buy.erc20(), req.sell
    }
```

Recommended Mitigation Steps

Consider having the affected code refactored as follows:

https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/Broker.sol#L97-L113

```diff
    function openTrade(TradeKind kind, TradeRequest memory req) 
-        require(!disabled, "broker disabled");

        address caller = _msgSender();
        require(
            caller == address(backingManager) ||
                caller == address(rsrTrader) ||
                caller == address(rTokenTrader),
            "only traders"
        );

        // Must be updated when new TradeKinds are created
        if (kind == TradeKind.BATCH_AUCTION) {
+            require(!disabled, "Gnosis Trade disabled");
            return newBatchAuction(req, caller);
        }
        return newDutchAuction(req, ITrading(caller));
    }
```

This will have the Gnosis Trade conditionally denied while still allowing the opening of Dutch Trade.

0xean (judge) commented:

> This currently mostly reads like a design suggestion. I can see the merits of disabling the entire broker in the scenario where the invariant has been violated. Probably best as QA, but will allow for sponsor comment before downgrading.

tbrent (Reserve) confirmed and commented:

> @0xean - We think this should be kept as Medium. It's a good design suggestion that otherwise could lead to the protocol not trading for the length of the governance cycle. This matters when it comes to selling defaulted collateral.

Reserve mitigated:

> Disable dutch auctions on a per-collateral basis, use 4-step dutch trade curve.
> PRs:

- https://github.com/reserve-protocol/protocol/pull/873
- https://github.com/reserve-protocol/protocol/pull/869

Status: Two mitigation errors. Full details in reports from ronnyx2017 and 0xA5DF - and also shared below in the Mitigation Review section.

[M-03] In case `Distributor.setDistribution` use, revenue from rToken RevenueTrader and rsr token RevenueTrader should be distributed

Submitted by rvierdiiev

In case Distributor.setDistribution use, revenue from rToken RevenueTrader and rsr token RevenueTrader should be distributed. Otherwise wrong distribution will be used.

Proof of Concept

`BackingManager.forwardRevenue` function sends revenue amount to the `rsrTrader` and `rTokenTrader` contracts, according to the distribution inside `Distributor` contract. For example it can `50%` / `50%` . In case if we have 2 destinations in Distributor: strsr and furnace, that means that half of revenue will be received by strsr stakers as rewards.

This distribution can be changed at any time.
The job of `RevenueTrader` is to sell provided token for a `tokenToBuy` and then distribute it using `Distributor.distribute` function. There are 2 ways of auction that are used: dutch and gnosis. Dutch auction will call `RevenueTrader.settleTrade` , which will initiate distribution. But Gnosis trade will not do that and user should call `distributeTokenToBuy` manually, after auction is settled.

The problem that I want to discuss is next.
Suppose, that governance at the beginning set distribution as 50/50 between 2 destinations: strsr and furnace. And then later `forwardRevenue` sent some tokens to the rsrTrader and rTokenTrader. Then, when trade was active to exchange some token to rsr token, `Distributor.setDistribution` was set in order to make strsr share to 0, so now everything goes to Furnace only. As result, when trade will be finished in the rsrTrader and `Distributor.distribute` will be called, then those tokens will not be sent to the strsr contract, because their share is 0 now. They will be stuck inside rsrTrader.

Another problem here is that strsr holders should receive all revenue from the time, where they distribution were created. What i mean is if in time 0, rsr share was 50% and in time 10 rsr share is 10%, then `BackingManager.forwardRevenue` should be called for all tokens that has surplus, because if that will be done after changing to 10%, then strsr stakers will receive less revenue.

Tools Used

VsCode

Recommended Mitigation Steps

You need to think how to guarantee fair distribution to the strsr stakers, when distribution params are changed.

tbrent (Reserve) confirmed and commented:

> This is a good find. The mitigation we have in mind is adding a new function to the `RevenueTrader` that allows anyone to transfer a registered ERC20 back to the `BackingManager`, as long as the current distribution for that `tokenToBuy` is 0%.

Reserve mitigated:

> Distribute revenue in `setDistribution`.
> PR: https://github.com/reserve-protocol/protocol/pull/878

Status: Mitigation error. Full details in reports from 0xA5DF and rvierdiiev - and also shared below in the Mitigation Review section.

[M-04] FurnaceP1.setRatio will work incorrectly after call when frozen

Submitted by rvierdiiev

`FurnaceP1.setRatio` will not update `lastPayout` when called in frozen state, which means that after component will be unfrozen, melting will be incorrect.

Proof of Concept

`melt` function should burn some amount of tokens from `lastPayoutBal`. It depends of `lastPayout` and `ratio` variables. The more time has passed, the more tokens will be burnt.

When `setRatio` function is called, then `melt` function is tried to be executed, because new ratio is provided and it should not be used for previous time ranges. In case if everything is ok, then `lastPayout` and `lastPayoutBal` will be updated, so it's safe to update `ratio` now. But it's possible that `melt` function will revert in case if `notFrozen` modifier is not passed. As result `lastPayout` and `lastPayoutBal` will not be updated, but ratio will be. Because of that, when `Furnace` will be unfrozen, then melting rate can be much more, then it should be, because `lastPayout` wasn't updated.

Tools Used

VsCode

Recommended Mitigation Steps

In case of `catch` case, you can update `lastPayout` and `lastPayoutBal`.

tbrent (Reserve) confirmed

Reserve mitigated:

> Update payout variables if melt fails during `setRatio`.
> PR: https://github.com/reserve-protocol/protocol/pull/885

Status: Mitigation error. Full details in report from 0xA5DF - and also shared below in the Mitigation Review section.

[M-05] Lack of claimRewards when manageToken in RevenueTrader

Submitted by ronnyx2017

There is a dev comment in the Assert.sol:

```
    DEPRECATED: claimRewards() will be removed from all assets and c
```

The claimRewards is moved to the `TradingP1.claimRewards/claimRewardsSingle`.

But when the `RevenueTraderP1` trade and distribute revenues by `manageToken`, it only calls the refresh function of the asserts:

```
    if (erc20 != IERC20(address(rToken)) && tokenToBuy != IERC20(add
        IAsset sell_ = assetRegistry.toAsset(erc20);
        IAsset buy_ = assetRegistry.toAsset(tokenToBuy);
        if (sell_.lastSave() != uint48(block.timestamp)) sell_.refre
        if (buy_.lastSave() != uint48(block.timestamp)) buy_.refresh
    }
```

The claimRewards is left out.

Impact

Potential loss of rewards.

Recommended Mitigation Steps

Add claimRewardsSingle when refresh assert in the `manageToken`.

[tbrent (Reserve) disputed and commented](https://github.com/code-423n4/2023-02-reserve-mitigation-contest-findings/issues/22):

> This is similar to an (unmitigated) issue from an earlier audit: https://github.com/code-423n4/2023-02-reserve-mitigation-contest-findings/issues/22

> However in this case it has to do with `RevenueTraderP1.manageToken()`, as opposed to `BackingManagerP1.manageTokens()`.

> I think that difference matters, because the loss of the rewards *for this auction* does not have serious long-term consequences. This is not like the BackingManager where it's important that all capital always be available else an unnecessarily large haircut could occur. Instead, the worst that can happen is for the revenue auction to complete at high slippage, and for a second reward token revenue auction to complete afterwards at high slippage yet again, when it could have been a single revenue auction with less slippage.

> The recommended mitigation would not succeed, because recall, we may be selling token X but any number of additional assets could have token X as a reward token. We would need to call `claimRewards()`, which is simply too gas-costly to do everytime for revenue auctions.

[0xean (judge) commented]():

> Instead, the worst that can happen is for the revenue auction to complete at high slippage, and for a second reward token revenue auction to complete afterwards at

> high slippage yet again, when it could have been a single revenue auction with less slippage.
> @tbrent - The impact sounds like a "leak of value" and therefore I think Medium is the correct severity per the c4 docs. (cc @tbrent - open to additional comment here)

[M-06] Oracle timeout at rebalance will result in a sell-off of all RSRs at 0 price

Submitted by ronnyx2017

When creating the trade for rebalance, the `RecollateralizationLibP1.nextTradePair` uses `(uint192 low, uint192 high) = rsrAsset.price(); // {UoA/tok}` to get the rsr sell price. And the rsr assert is a pure Assert contract, which `price()` function will just return (0, FIX_MAX) if oracle is timeout:

```
    function price() public view virtual returns (uint192, uint192)
        try this.tryPrice() returns (uint192 low, uint192 high, uint
            assert(low <= high);
            return (low, high);
        } catch (bytes memory errData) {
            ...
            return (0, FIX_MAX);
        }
    }
```

The `trade.sellAmount` will be all the rsr in the `BackingManager` and `stRSR`:

```
    uint192 rsrAvailable = rsrAsset.bal(address(ctx.bm)).plus(
        rsrAsset.bal(address(ctx.stRSR))
    );
    trade.sellAmount = rsrAvailable;
```

It will be cut down to a normal amount fit for buying UoA amount in the `trade.prepareTradeToCoverDeficit` function.

But if the rsr oracle is timeout and returns a 0 low price. The trade req will be made by `trade.prepareTradeSell`, which will sell all the available rsr at 0 price.

Note that the SOUND colls won't be affected by the issue because the sell amount has already been cut down by basketsNeeded.

Loss huge amount of rsr in the auction. When huge amounts of assets are auctioned off at zero, panic and insufficient liquidity make the outcome unpredictable.

Proof of Concept

POC git diff test/Recollateralization.test.ts

```
diff --git a/test/Recollateralization.test.ts b/test/Recollatera
index 86cd3e88..15639916 100644
--- a/test/Recollateralization.test.ts
+++ b/test/Recollateralization.test.ts
@@ -51,7 +51,7 @@ import {
 import snapshotGasCost from './utils/snapshotGasCost'
 import { expectTrade, getTrade, dutchBuyAmount } from './utils/
 import { withinQuad } from './utils/matchers'
-import { expectRTokenPrice, setOraclePrice } from './utils/orac
+import { expectRTokenPrice, setInvalidOracleTimestamp, setOracl
 import { useEnv } from '#/utils/env'
 import { mintCollaterals } from './utils/tokens'

@@ -797,6 +797,166 @@ describe(`Recollateralization - P${IMPLEME
   })

   describe('Recollateralization', function () {
+    context('With simple Basket - Two stablecoins', function ()
+      let issueAmount: BigNumber
+      let stakeAmount: BigNumber
+
+      beforeEach(async function () {
+        // Issue some RTokens to user
+        issueAmount = bn('100e18')
+        stakeAmount = bn('10000e18')
+
+        // Setup new basket with token0 & token1
+        await basketHandler.connect(owner).setPrimeBasket([toke
+        await basketHandler.connect(owner).refreshBasket()
+
+         // Provide approvals
+         await token0.connect(addr1).approve(rToken.address, ini
+         await token1.connect(addr1).approve(rToken.address, ini
+
+         // Issue rTokens
+         await rToken.connect(addr1).issue(issueAmount)
+
+         // Stake some RSR
+         await rsr.connect(owner).mint(addr1.address, initialBal
+         await rsr.connect(addr1).approve(stRSR.address, stakeAm
+         await stRSR.connect(addr1).stake(stakeAmount)
+       })
+
+       it('C4M7', async () => {
+         // Register Collateral
+         await assetRegistry.connect(owner).register(backupColla
+
+         // Set backup configuration - USDT as backup
+         await basketHandler
+           .connect(owner)
+           .setBackupConfig(ethers.utils.formatBytes32String('US
+
+         // Set Token0 to default - 50% price reduction
+         await setOraclePrice(collateral0.address, bn('0.5e8'))
+
+         // Mark default as probable
+         await assetRegistry.refresh()
+         // Advance time post collateral's default delay
+         await advanceTime((await collateral0.delayUntilDefault(
+
+         // Confirm default and trigger basket switch
+         await basketHandler.refreshBasket()
+
+         // Advance time post warmup period - SOUND just regaine
+         await advanceTime(Number(config.warmupPeriod) + 1)
+
+         const initToken1B = await token1.balanceOf(backingManag
+         // rebalance
+         const token1Decimal = 6;
+         const sellAmt: BigNumber = await token0.balanceOf(backi
+         const buyAmt: BigNumber = sellAmt.div(2)
+         await facadeTest.runAuctionsForAllTraders(rToken.addres
+         // bid
+         await backupToken1.connect(addr1).approve(gnosis.addres
+         await gnosis.placeBid(0, {
+           bidder: addr1.address,
+           sellAmount: sellAmt,
+           buyAmount: buyAmt,
+         })
+         await advanceTime(config.batchAuctionLength.add(100).to
+         // await facadeTest.runAuctionsForAllTraders(rToken.add
+         const rsrAssert = await assetRegistry.callStatic.toAsse
+         await setInvalidOracleTimestamp(rsrAssert);
+         await expectEvents(facadeTest.runAuctionsForAllTraders(
+           {
+             contract: backingManager,
+             name: 'TradeSettled',
+             args: [anyValue, token0.address, backupToken1.addre
+             emitted: true,
+           },
+           {
+             contract: backingManager,
+             name: 'TradeStarted',
+             args: [anyValue, rsr.address, backupToken1.address,
+             emitted: true,
+           },
+         ])
+
+         // check
+         console.log(await token0.balanceOf(backingManager.addre
+         const currentToken1B = await token1.balanceOf(backingMa
+         console.log(currentToken1B);
+         console.log(await backupToken1.balanceOf(backingManager
+         const rsrB = await rsr.balanceOf(stRSR.address);
+         console.log(rsrB);
+
+         // expect
+         expect(rsrB).to.eq(0);
+       })
+     })
+
      context('With very simple Basket - Single stablecoin', func
        let issueAmount: BigNumber
        let stakeAmount: BigNumber
```

run test:

```
PROTO_IMPL=1 npx hardhat test --grep 'C4M7' test/Recollateraliza
```

log:

```
      Recollateralization - P1
        Recollateralization
          With simple Basket - Two stablecoins
    BigNumber { value: "0" }
    BigNumber { value: "50000000" }
    BigNumber { value: "25000000000000000000" }
    BigNumber { value: "0" }
```

Recommended Mitigation Steps

Using lotPrice or just revert for rsr oracle timeout might be a good idea.

tbrent (Reserve) confirmed and commented:

> Hmm, interesting case.
>
> There are two types of auctions that can occur: batch auctions via `GnosisTrade`, and dutch auctions via `DutchTrade`.
>
> Batch auctions via `GnosisTrade` are good at discovering prices when price is unknown. It would require self-interested parties to be offline for the entire duration of the batch auction (default: 15 minutes) in order for someone to get away with buying the RSR for close to 0.
>
> Dutch auctions via `DutchTrade` do not have this problem because of an assert that reverts at the top of the contract.
>
> I'm inclined to dispute validity, but I also agree it might be strictly better to use the `lotPrice()`. When trading out backing collateral it is important to sell it quickly and not have to wait for `lotPrice()` to decay sufficiently, but this is not true with RSR. For RSR it might be fine to wait as long as a week for the `lotPrice()` to fall to near 0.
>
> This would then allow dutch auctions via `DutchTrade` to be used when RSR's oracle is offline.

Reserve mitigated:

> Use `lotPrice()`.
> PR: https://github.com/reserve-protocol/protocol-private/pull/15

Status: Mitigation confirmed. Full details in reports from rvierdiiev, 0xA5DF, and ronnyx2017 - and also shared below in the Mitigation Review section.

[M-07] Sell reward `rTokens` at low price because of skiping `furnace.melt`

Submitted by ronnyx2017

The reward rToken sent to RevenueTrader will be sold at a low price. RSR stakers will lose some of their profits.

Proof of Concept

`RevenueTraderP1.manageToken` function is used to launch auctions for any erc20 tokens sent to it. For the RevenueTrader of the rsr stake, the `tokenToBuy` is rsr and the token to sell is reward rtoken.

There is the refresh code in the `manageToken` function:

```
    } else if (assetRegistry.lastRefresh() != uint48(block.timestamp
        // Refresh everything only if RToken is being traded
        assetRegistry.refresh();
        furnace.melt();
    }
```

It refreshes only when the assetRegistry has not been refreshed in the same block.

So if the actor calls the `assetRegistry.refresh()` before calling `manageToken` function, the `furnace.melt()` won't been called. And the BU exchange rate of the RToken will be lower than actual value. So the sellPrice is also going to be smaller.

```
    (uint192 sellPrice, ) = sell.price(); // {UoA/tok}

    TradeInfo memory trade = TradeInfo({
        sell: sell,
        buy: buy,
        sellAmount: sell.bal(address(this)),
        buyAmount: 0,
        sellPrice: sellPrice,
        buyPrice: buyPrice
    });
```

Recommended Mitigation Steps

Refresh everything before sell rewards.

tbrent (Reserve) confirmed

Reserve mitigated:

> Refresh before selling rewards, refactor revenue & distro.
> PR: https://github.com/reserve-protocol/protocol-private/pull/7

Status: Mitigation confirmed. Full details in reports from ronnyx2017, 0xA5DF, and rvierdiiev - and also shared below in the Mitigation Review section.

[M-08] Stake before unfreeze can take away most of rsr rewards in the freeze period

Submitted by ronnyx2017, *also found by* rvierdiiev *and* 0xA5DF

If the system is frozen, the only allowed operation is `stRST.stake`. And the `_payoutRewards` is not called during freeze period:

```
    if (!main.frozen()) _payoutRewards();

    function payoutRewards() external {
        requireNotFrozen();
        _payoutRewards();
    }
```

So the `payoutLastPaid` stays before the freeze period. But when the system is unfreezed, accumulated rewards will be released all at once because the block.timestamp leapt the whole freeze period.

Impact

A front runner can stake huge proportion rsr before admin unfreezes the system. And the attacker can get most of rsr rewards in the next block. And he only takes the risk of the `unstakingDelay` period.

Proof of Concept

Assumption: there are 2000 rsr stake in the stRSR, and there are 1000 rsr rewards in the `rsrRewardsAtLastPayout` with a 1 year half-life period.

And at present, the LONG_FREEZER `freezeLong` system for 1 year(default).

After 1 year, at the unfreeze point, a front runner stake 2000 rsr into stRSR. And then the system is unfreeze. And in the next blcok,the front runner unstakes all the stRSR he has for `2250 rsr = 2000 principal + 1000 / 2 / 2 rsr rewards`.

The only risk he took is `unstakingDelay`. The original rsr stakers took the risk of the whole freeze period + `unstakingDelay` but only got a part of rewards back.

Recommended Mitigation Steps

payoutRewards before freeze and update payoutLastPaid before unfreeze.

[tbrent (Reserve) confirmed via duplicate issue #24](#)

[Reserve mitigated](#):

> `payoutRewards` before freeze and update `payoutLastPaid` before unfreeze.
> PR: https://github.com/reserve-protocol/protocol/pull/857

Status: Mitigation confirmed. Full details in reports from [rvierdiiev](#), [0xA5DF](#), and [ronnyx2017](#) - and also shared below in the [Mitigation Review](#) section.

[M-09] `cancelUnstake` lack `payoutRewards` before mint shares

Submitted by ronnyx2017, also found by rvierdiiev and 0xA5DF

`cancelUnstake` will cancel the withdrawal request in the queue can mint shares as the current `stakeRate`. But it doesn't `payoutRewards` before `mintStakes`. Therefor it will mint stRsr as a lower rate, which means it will get more rsr.

Impact

Withdrawers in the unstake queue can `cancelUnstake` without calling `payoutRewards` to get more rsr rewards that should not belong to them.

Proof of Concept

POC test/ZZStRSR.test.ts git patch

```
diff --git a/test/ZZStRSR.test.ts b/test/ZZStRSR.test.ts
index ecc31f68..b2809129 100644
--- a/test/ZZStRSR.test.ts
+++ b/test/ZZStRSR.test.ts
@@ -1333,6 +1333,46 @@ describe(`StRSRP${IMPLEMENTATION} contrac
       expect(await stRSR.exchangeRate()).to.be.gt(initialRate)
     })

+    it('cancelUnstake', async () => {
+      const amount: BigNumber = bn('10e18')
+
+      // Stake
+      await rsr.connect(addr1).approve(stRSR.address, amount)
+      await stRSR.connect(addr1).stake(amount)
+      await rsr.connect(addr2).approve(stRSR.address, amount)
+      await stRSR.connect(addr2).stake(amount)
+      await rsr.connect(addr3).approve(stRSR.address, amount)
+      await stRSR.connect(addr3).stake(amount)
+
+      const  initExchangeRate = await stRSR.exchangeRate();
+      console.log(initExchangeRate);
+
+      // Unstake addr2 & addr3 at same time (Although in differ
+       await stRSR.connect(addr2).unstake(amount)
+       await stRSR.connect(addr3).unstake(amount)
+
+       // skip 1000 block PERIOD / 12000s
+       await setNextBlockTimestamp(Number(ONE_PERIOD.mul(1000).ad
+
+       // Let's cancel the unstake in normal
+       await expect(stRSR.connect(addr2).cancelUnstake(1)).to.em
+       let exchangeRate = await stRSR.exchangeRate();
+       expect(exchangeRate).to.equal(initExchangeRate)
+
+       // addr3 cancelUnstake after payoutRewards
+       await stRSR.payoutRewards()
+       await expect(stRSR.connect(addr3).cancelUnstake(1)).to.em
+
+       // Check balances addr2 & addr3
+       exchangeRate = await stRSR.exchangeRate();
+       expect(exchangeRate).to.be.gt(initExchangeRate)
+       const addr2NowAmount = exchangeRate.mul(await stRSR.balan
+       console.log("addr2", addr2NowAmount.toString());
+       const addr3NowAmount = exchangeRate.mul(await stRSR.balan
+       console.log("addr3",addr3NowAmount.toString());
+       expect(addr2NowAmount).to.gt(addr3NowAmount)
+     })
+
      it('Rewards should not be handed out when paused but stakin
        await main.connect(owner).pauseTrading()
        await setNextBlockTimestamp(Number(ONE_PERIOD.add(await ge
```

The test simulates two users unstake and cancelUnstake operations at the same time.But the addr2 calls payoutRewards after his cancelUnstake. And addr3 calls cancelUnstake after payoutRewards. Addr2 gets more rsr than addr3 in the end.

run test:

```
PROTO_IMPL=1 npx hardhat test --grep cancelUnstake test/ZZStRSR.t
```

log:

```
  StRSRP1 contract
    Add RSR / Rewards
BigNumber { value: "1000000000000000000" }
addr2 1000534550125858824O
addr3 1000000000000000013
```

Recommended Mitigation Steps

Call `_payoutRewards` before mint shares.

tbrent (Reserve) confirmed and commented:

> Agree with severity and proposed mitigation.

Reserve mitigated:

> Payout rewards during cancelUnstake.
> PR: https://github.com/reserve-protocol/protocol-private/pull/3

Status: Mitigation confirmed. Full details in reports from rvierdiiev, 0xA5DF, and ronnyx2017 - and also shared below in the Mitigation Review section.

[M-10] An oracle deprecation might lead the protocol to sell assets for a low price

Submitted by 0xA5DF

During a Dutch Auction, if a user places a bid, the trade is settled in the same transaction. As part of this process, the backing manager tries to call the `rebalance()` function again. The call to `rebalance()` is wrapped in a try-catch block, if an error occurs and the error data is empty, the function will revert.

The assumption is that if the error data is empty that means it was due to an out-of-gas error, this assumption isn't always true as mentioned in a previous issue (that wasn't mitigated). In the case of this issue, this can result in a case where users can't bid on an auction for some time, ending up selling an asset for a price lower than the market price.

Impact

Protocol's assets will be auctioned for a price lower than the market price.

Proof of Concept

Consider the following scenario:

- Chainlink announces that an oracle will get deprecated
- Governance passes a proposal to update the asset registry with a new oracle
- A re-balancing is required and executed with a Dutch Auction
- The oracle deprecation happens before the auction price reaches a reasonable value
- Any bid while the oracle is deprecated will revert
- Right before the auction ends the proposal to update the asset becomes available for execution (after the timelock delay has passed). Somebody executes it, bids, and enjoys the low price of the auction.

Recommended Mitigation Steps

On top of checking that the error data is empty, compare the gas before and after to ensure this is an out-of-gas error.

0xean (judge) commented:

> On the fence on this one, it is based off a known issue from a previous Audit but does show a new problem stemming from the same problem of oracle deprecation.

> Look forward to sponsor comment.

tbrent (Reserve) disputed and commented:

> The PoC does not function as specified. Specifically, bidding on an auction does not involve the price at the time of the tx. The price is set at the beginning of the dutch auction in the `init()` function. Therefore, it is the starting of new auctions that will revert while the oracle is deprecated, while bids will succeed and simply fail to start the next auction.

0xA5DF (warden) commented:

> Therefore, it is the starting of new auctions that will revert while the oracle is deprecated, while bids will succeed and simply fail to start the next auction.

> Hey @tbrent - I didn't quite understand the dispute here, if starting the next auction will fail/revert then the bid will revert too.
> `bid()` calls `origin.settleTrade()` and `settleTrade()` calls `rebalance()`.
> If `rebalance()` reverts due to a deprecated oracle then `settleTrade()` will revert too (`rebalance()` will revert with empty data, and therefore the catch block will trigger a revert here).

tbrent (Reserve) confirmed and commented:

> @0xA5DF - Ah, understood now. Agree this is Medium and think it should be counted as a new finding since the consequence (dutch auction economics break) is novel.

tbrent (Reserve) commented:

> Hey @0xa5df — we're having some confusion around exactly what happens when a chainlink oracle is deprecated. Do you have details to share about what this ends up looking like?

> We're having trouble finding documentation on this, and it feels like the aggregator contract should just stay there and return a stale value. Is that not right? Has this happened in the past or has Chainlink committed to a particular approach for deprecating?

0xA5DF (warden) commented:

> Hey - It's a bit difficult to track deprecated Chainlink oracles since Chainlink removes the announcement once they're deprecated.
> I was able to track one Oracle that was deprecated during the first audit, from the original issue this seems to be this one.
> It seems that what happens is that Chainlink sets the aggregator address to the zero address, which makes the call to `latestRoundData()` to revert without any data (I guess this is due to the way Solidity handles calls to a non-contract address).
> See also the PoC in the original issue in the January audit.

tbrent (Reserve) commented:

> Got it, checks out. Thanks!

Reserve mitigated:

> Add oracle deprecation check.
> PR: https://github.com/reserve-protocol/protocol/pull/886

Status: Mitigation confirmed. Full details in reports from 0xA5DF, ronnyx2017, and rvierdiiev - and also shared below in the Mitigation Review section.

[M-11] Attacker can disable basket during un-registration, which can cause an unnecessary trade in some cases

Submitted by 0xA5DF*, also found by* rvierdiiev

https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/AssetRegistry.sol#L89-L93
https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/AssetRegistry.sol#L106-L110

At the mitigation audit there was an issue regarding the `basketHandler.quantity()` call at the unregistration process taking up all gas. As a mitigation to that issue the devs set aside some gas and use the remaining to do that call. This opens up to a new kind of attack, where a attacker can cause the call to revert by not supplying enough gas to it.

Impact

This can cause the basket to get disabled, which would require a basket refresh.

After a basket refresh is done, an additional warmup period has to pass for some functionality to be available again (issuance, rebalancing and forwarding revenue).

In some cases this might trigger a basket switch that would require the protocol to rebalance via trading, trading can have some slippage which can cause a loss for the protocol.

Proof of Concept

The `quantity()` function is being called with the amount of gas that `_reserveGas()` returns

If an attacker causes the gas to be just right above `GAS_TO_RESERVE` the function would be called with 1 unit of gas, causing it to revert:

```
    function _reserveGas() private view returns (uint256) {
        uint256 gas = gasleft();
        require(gas > GAS_TO_RESERVE, "not enough gas to unregis
        return gas - GAS_TO_RESERVE;
    }
```

Regarding the unnecessary trade, consider the following scenario:

- The basket has USDC as the main asset and DAI as a backup token
- A proposal to replace the backup token with USDT was raised
- A proposal to unregister BUSD (which isn't part of the basket) was raised too
- USDC defaults and DAI kicks in as the backup token
- Both proposals are now ready to execute and the attacker executes the backup proposal first, then the unregister while disabling the basket using the bug in question
- Now, when the basket is refreshed DAI will be replaced with USDT, making the protocol to trade DAI for USDT

The refresh was unnecessary and therefore the trade too.

Recommended Mitigation Steps

Reserve gas for the call as well:

```
    function _reserveGas() private view returns (uint256) {
            uint256 gas = gasleft();
-           require(gas > GAS_TO_RESERVE, "not enough gas to unregi
+           require(gas >= GAS_TO_RESERVE + MIN_GAS_FOR_EXECUTION,
            return gas - GAS_TO_RESERVE;
        }
```

Disclosure: this issue was [mentioned in the comments]() to the issue in the mitigation audit; however, since this wasn't noticed by the devs and isn't part of the submission, I don't think this should be considered a known issue.

[0xean (judge) commented]():

> Applaud @0xA5DF for highlighting this on their own issue.

> *Disclosure: this issue was [mentioned in the comments]() to the issue in the mitigation audit, however since this wasn't noticed by the devs and isn't part of the submission I don't think this should be considered a known issue*

> Look forward to discussion with sponsor.

[tbrent (Reserve) confirmed and commented]():

> We've discussed and agree with with the warden that this should not be considered a known issue.

[Reserve mitigated]():

> Change gas reservation policy in `AssetRegistry`.
> PR: https://github.com/reserve-protocol/protocol/pull/857

Status: Mitigation confirmed. Full details in reports from [ronnyx2017](), [0xA5DF](), and [rvierdiiev]() - and also shared below in the [Mitigation Review]() section.

[M-12] Custom redemption can be used to get more than RToken value, when an upwards depeg occurs

Submitted by [0xA5DF]()

https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/RToken.sol#L245-L338
https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/BasketHandler.sol#L437-L440

Custom redemption allows to redeem RToken in exchange of a mix of previous baskets (as long as it's not more than the prorata share of the redeemer). The assumption is that previous baskets aren't worth more than the target value of the basket. However, a previous basket can contain a collateral that depegged upwards and is now actually worth more than its target.

Impact

Funds that are supposed to go revenue traders would be taken by an attacker redeeming RToken.

Proof of Concept

The following code shows that when a depeg occurs the collateral becomes IFFY (which means it'll be disabled after a certain delay):

```
                // If the price is below the default-threshold price
                // uint192(+/-) is the same as Fix.plus/minus
                if (pegPrice < pegBottom || pegPrice > pegTop || low
                    markStatus(CollateralStatus.IFFY);
                } else {
                    markStatus(CollateralStatus.SOUND);
                }
```

Consider the following scenario:

- Basket contains token X which is supposed to be pegged to c
- Token X depegs upwards and is now worth 2c
- After `delayUntilDefault` passes the basket gets disabled
- A basket refresh is executed (can be done by anybody) and token Y kicks in as the backup token

- Half of token X is now traded for the required Y tokens
- The other half should go to revenue traders (rsr trader and Furnace), but before anyone calls 'forewardRevenue' the attacker calls custom redemption with half from the current basket and half of the previous one
- The user would get 0.5X+0.5Y per each RToken which is worth 1.5c

Recommended Mitigation Steps

When doing custom redemption check that the collateral used is sound or at least check that the price isn't higher then the peg price.

tbrent (Reserve) acknowledged, but disagreed with severity and commented:

> This is also true of normal redemption via `redeem()` until `refreshBasket()` is called after the collateral has cycled from IFFY to DISABLED.

> It only checks `basketHandler.fullyCollateralized()`, not `basketHandler.isReady()`. This is intended. It is important to not disallow redemption, and using USD prices to determine redemption quantities is opposite to the fundamental design of the protocol.

> Agree that the behavior is as the warden indicates. All alternatives seem worse, however. I think this is probably not HIGH. We do not expect to make any change to the behavior.

0xean (judge) reduced severity to Medium and commented:

> I think Medium seems like the correct severity here. There are pre-conditions for this to occur and in addition the likelihood doesn't seem very high.

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 **0xA5DF** received the top score from the judge.

The following wardens also submitted reports: hihen, RaymondFam, rvierdiiev, carlitox477, *and* ronnyx2017.

[01] A redeemer might get ‘front-run’ by a freezer

Before redemption `furnace.melt()` is called, which increases a bit the amount of assets the redeemer gets in return. While frozen the `melt()` call will revert, but since the call is surrounded by a try-catch block the redemption will continue.

This can lead to a case where a user sends out a redeem tx, expecting melt to be called by the redeem function, but before the tx is executed the protocol gets frozen. This means the user wouldn’t get the additional value they expected to get from melting (and that they can get if they choose to wait till after the freeze is over).

If the last time `melt()` was called wasn’t long enough then the additional value from melting wouldn’t be that significant. But there can be cases where it can be longer - e.g. low activity or after a freeze (meaning there are 2 freezes one after the other and a user is attempting to redeem between them).

As a mitigation allow the user to specify if they’re ok with skipping the call to `melt()`, if they didn’t specifically allow it then revert.

[02] Leaky refresh math is incorrect

`leakyRefresh()` keeps track of the percentage that was withdrawn since last refresh by adding up the current percentage that’s being withdrawn each time. However, the current percentage is being calculated as part of the current `totalRSR`, which doesn’t account for the already withdrawn drafts.

This will trigger the `withdrawalLeak` threshold earlier than expected. E.g. if the threshold is set to `25%` and 26 users try to withdraw `1%` each - the leaky refresh would be triggered by the 23rd person rather than by the 26th.

[03] Reorg attack

Reorg attacks aren’t very common on mainnet (but more common on L2s if the protocol intends to ever launch on them), but they can still happen (there was a 7 blocks reorg on the Beacon chain before the merge). It can be relevant in the following cases (I’ve reported a med separately, the followings are the ones I consider low):

- RToken deployment - a user might mint right after the RToken was deployed, a reorg might be used to deploy a different RToken and trap the users' funds in it (since the deployer becomes the owner).
- Dutch Auction - a reorg might switch the addresses of 2 auctions, causing the user to bid on the wrong auction
- Gnosis auctions - this can also cause the user to bid on the wrong auction (it's more relevant for the `EasyAuction` contract which is OOS)

As a mitigation - make sure to deploy all contracts with `create2` using a salt that's unique to the features of the contract, that will ensure that even in the case of a reorg it wouldn't be deployed to the same address as before.

[04] `distributeTokenToBuy()` can be called while paused/frozen

Due to the removal of `notPausedOrFrozen` from `Distributor.distribute()` it's now possible to execute `RevenueTrader.distributeTokenToBuy()` while paused or frozen.

This is relevant when `tokensToBuy` is sent directly to the revenue trader: RSR sent to RSRTrader or RToken sent directly to RTokenTrader

[05] `redeemCustom` allows the use of the zero basket

The basket with nonce `#0` is an empty basket, `redeemCustom` allows to specify that basket for redemption, which will result in a loss for the redeemer.

[06] `refreshBasket` can be called before the first prime basket was set

This will result in an event being emitted but will not impact the contract's state.

[07] `MIN_AUCTION_LENGTH` seems too low

The current `MIN_AUCTION_LENGTH` is set to 2 blocks. This seems a bit too low since the price is time-dependant that means there would be only 3 price options for the auctions, and the final price wouldn't necessarily be the optimal price for the protocol.

[08] If a token that yields RSR would be used as collateral then 100% of the yield would go to StRSR

This isn't very likely to happen (currently the only token that yields RSR is StRSR of another RToken) but it's worth keeping an eye on it.

[09] Protocol might not be able to compromise basket when needed

Consider the following scenario:

- Protocol suffers from some loss and compromises the basket to a 1.1e9 ratio
- Months pass by and users mint new tokens and increase the TVL
- A small compromise is required (12%), this brings the ratio to below 1e9 and reverts the compromise
- Protocol is now disabled despite holding a significant amount of value, and users can only redeem for their prorata share of the assets,

This might be intended design, but worth taking this scenario into account.

Gas Optimizations

For this Audit, 4 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **0xA5DF** received the top score from the judge.

The following wardens also submitted reports: hihen, RaymondFam, *and* carlitox477.

[G-01] At `toColl()` and `toAsset()` use the mapping to check if asset exists

Savings: ~2.1K per call

Notice this is a function that's being called frequently, and many times per tx.

Overall this can save a few thousands of gas units per tx for the most common txs (e.g. issuance, redemption)

https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/AssetRegistry.sol#L120-L132

At `toColl()` and `toAsset()` instead of using the EnumberableSet to check that the erc20 is registered just check that the value returned from the mapping isn't zero (this is supposed to be equivalent as long as the governance doesn't register the zero address as an asset contract - maybe add a check for that at `register()`).

Proposed changes:

```
    /// Return the Asset registered for erc20; revert if erc20 i
    // checks: erc20 in assets
    // returns: assets[erc20]
    function toAsset(IERC20 erc20) external view returns (IAsset
        IAsset asset = assets[erc20];
        require(asset != IAsset(address(0)), "erc20 unregistered
        return asset;
    }

    /// Return the Collateral registered for erc20; revert if er
    // checks: erc20 in assets, assets[erc20].isCollateral()
    // returns: assets[erc20]
    function toColl(IERC20 erc20) external view returns (ICollat
        IAsset coll = assets[erc20];
        require(coll != IAsset(address(0)), "erc20 unregistered"
        require(coll.isCollateral(), "erc20 is not collateral");
        return ICollateral(address(coll));
    }
```

[G-02] Get `targetIndex` from mapping instead of iterating

Gas savings: a few thousands (see below)

The following code is used at the BasketLib to find the index of a value inside an `EnumerableSet`

```
            for (targetIndex = 0; targetIndex < targetsLength; +
                if (targetNames.at(targetIndex) == config.targetN
                }
```

However the index can be fetched directly from the `_indexed` mapping:

```
diff --git a/contracts/p1/mixins/BasketLib.sol b/contracts/p1/mi
index bc52d1c6..ce56c715 100644
--- a/contracts/p1/mixins/BasketLib.sol
+++ b/contracts/p1/mixins/BasketLib.sol
@@ -192,10 +192,8 @@ library BasketLibP1 {
         // For each prime collateral token:
         for (uint256 i = 0; i < config.erc20s.length; ++i) {
             // Find collateral's targetName index
-            uint256 targetIndex;
-            for (targetIndex = 0; targetIndex < targetsLength;
-                if (targetNames.at(targetIndex) == config.targe
-            }
+            uint256 targetIndex = targetNames._inner._indexes[c
+
             assert(targetIndex < targetsLength);
             // now, targetNames[targetIndex] == config.targetNa
```

Gas savings:

- The `_indexes` keys are considered warm since all values were inserted in the current tx
- Total saving is the sum of the index of the target names per each erc20 minus 1
- on average (depends on the location of the target in the set for each erc20): `(config.erc20s.length)*(targetsLength-1)/2*100`
    - E.g. for target length of 5 and 10 ERC20 that would save on average `10*4/2*100=2K`

[G-03] Use `furnace` instead of `main.furnace()`

Gas savings: ~2.6K

Code: https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/RToken.sol#L184 https://github.com/reserve-protocol/protocol/blob/c4ec2473bbcb4831d62af55d275368e73e16b984/contracts/p1/RToken.sol#L257

At `RToken.redeemTo()` and `redeemCustom()` furnace is being called using `main.furnace()` instead of using the `furnace` variable.

The call to `main.furnace()` costs both the cold storage variable read at `main.furnace()` and an external call to a cold address while using the `furnace` variable of the current contract costs only the cold storage read.

The additional cold call would cost ~2.6K.

To my understanding both of the values should be equal at all times so there shouldn't be an issue with the replacement.

[G-04] Deployer.implementations can be immutable

Gas saved: ~28K per RToken deployment

The struct itself can't be immutable, but you can save the values of the fields (and fields of the `components`) as immutable variables, and use an internal function to build the struct out of those immutable variables.

This would save ~2.1K per field, with 13 fields that brings us to ~28K of units saved.

[G-05] Update `lastWithdrawRefresh` only if it has changed

Gas saved: ~100

At `leakyRefresh()` `lastWithdrawRefresh` gets updated even if didn't change, that costs an additional 100 gas units.

Proposed change:

```
-           leaked = lastWithdrawRefresh != lastRefresh ? withdrawa
-           lastWithdrawRefresh = lastRefresh;
+           if(lastWithdrawRefresh != lastRefresh){
```

```
+                    leaked =  withdrawal;
+                    lastWithdrawRefresh = lastRefresh;
+            } else{
+                    leaked = leaked + withdrawal;
+            }
```

[G-06] Require array to be sorted and use `sortedAndAllUnique` at `BackingManager.forwardRevenue()`

Estimated savings: `~n^2*10` where n is the length of the asset.

For example for 20 assets that would save ~4K.

[G-07] Caching storage variable and function calls

This is one of the most common ways to save a nice amount of gas. Every additional read costs 100 gas units (when it comes to mapping or arrays there's additional cost), and each additional function call costs at least 100 gas units (usually much more).

I've noticed a few instances where a storage variable read or a view-function call can be cached to memory to save gas, I'm pretty sure there are many more instances that I didn't notice.

[G-08] `BasketLib.nextBasket()` refactoring

Gas saved: a few thousand

The following refactoring saves a few thousands of gas mostly by preventing:

1. Double call to `goodCollateral`
2. The second iteration over the whole `backup.erc20s` array

```
diff --git a/contracts/p1/mixins/BasketLib.sol b/contracts/p1/mixins/Ba
index bc52d1c6..7ab9c48b 100644
--- a/contracts/p1/mixins/BasketLib.sol
+++ b/contracts/p1/mixins/BasketLib.sol
@@ -192,10 +192,8 @@ library BasketLibP1 {
     // For each prime collateral token:
     for (uint256 i = 0; i < config.erc20s.length; ++i) {
         // Find collateral's targetName index
```

3. uint256 targetIndex;
4. for (targetIndex = 0; targetIndex < targetsLength; ++targetIndex) {
5. if (targetNames.at(targetIndex) == config.targetNames[config.erc20s[i]]) break;
6. }
7. uint256 targetIndex = targetNames.*inner*.indexes[config.targetNames[config.erc20s[i]]] -1 ;

8.
```
            assert(targetIndex < targetsLength);
            // now, targetNames[targetIndex] == config.targetNames[erc20]
```

@@ -244,32 +242,32 @@ library BasketLibP1 { uint256 size = 0; // backup basket size BackupConfig storage backup = config.backups[targetNames.at(i)];

- IERC20[] memory backupsToUse = new IERC20;
- + // Find the backup basket size: min(backup.max, # of good backup collateral) for (uint256 j = 0; j < backup.erc20s.length && size < backup.max; ++j) {
- if (goodCollateral(targetNames.at(i), backup.erc20s[j], assetRegistry)) size++;
- if (goodCollateral(targetNames.at(i), backup.erc20s[j], assetRegistry))
- {
- backupsToUse[size] = backup.erc20s[j];
- size++;
- } }

```
            // Now, size = len(backups(tgt)). If empty, fail.
            if (size == 0) return false;
```

- // Set backup basket weights...
- uint256 assigned = 0;

```
            // Loop: for erc20 in backups(tgt)...
```

- for (uint256 j = 0; j < backup.erc20s.length && assigned < size; ++j) {

- if (goodCollateral(targetNames.at(i), backup.erc20s[j], assetRegistry)) {
- // Across this .add(), targetWeight(newBasket',erc20)
- // = targetWeight(newBasket,erc20) + unsoundPrimeWt(tgt) / len(backups(tgt))
- newBasket.add(
- backup.erc20s[j],
- totalWeights[i].minus(goodWeights[i]).div(
- // this div is safe: targetPerRef > 0: goodCollateral check
- assetRegistry.toColl(backup.erc20s[j]).targetPerRef().mulu(size),
- CEIL
-)
-);
- assigned++;
- }
- for (uint256 j = 0; j < size; ++j) {
- +
- newBasket.add(
- backupsToUse[j],
- totalWeights[i].minus(goodWeights[i]).div(
- // this div is safe: targetPerRef > 0: goodCollateral check
- assetRegistry.toColl(backupsToUse[j]).targetPerRef().mulu(size),
- CEIL
-)
-); } // Here, targetWeight(newBasket, e) = primeWt(e) + backupWt(e) for all e targeting tgt }

[G-09] `BasketLib.nextBasket()` caching

On top of the above refactoring:

- `config.erc20s[i]` is being read a few times here
- `config.erc20s.length` and `backup.erc20s.length` can be cached
- `targetNames.at(i)` is being read twice in the second loop (3 before the proposed refactoring)

[G-10] `sellAmount` at `DutchTrade.settle()`

`sellAmount` is read here twice if greater than `sellBal`

```
            soldAmt = sellAmount > sellBal ? sellAmount - sellBal :
```

[G-11] `quoteCustomRedemption()` loop

`nonce`

Savings: `100*basketNonces.length`

Here nonce is being read each iteration. It can be cached outside of the loop.

`basketNonces.length`

Savings: `100*basketNonces.length`

`b.erc20s.length`

Savings: `100*(sum of` b.erc20s.length `for all baskets)`

[G-12] Use custom errors instead of string errors

This saves gas both for deployment and in case that the revert is triggered.

Mitigation Review

Introduction

Following the C4 audit, 3 wardens (0xA5DF, ronnyx2017, and rvierdiiev) reviewed the mitigations for all identified issues. Additional details can be found within the C4 Reserve Protocol Mitigation Review repository.

Mitigation Review Scope

Branch

https://github.com/reserve-protocol/protocol/pull/882 (commit hash 99d9db72e04db29f8e80e50a78b16a0b475d79f3)

Individual PRs

URL	Mitigation of	Purpose
https://github.com/reserve-protocol/protocol/pull/857	H-01	Fix redeemCustom
https://github.com/reserve-protocol/protocol/pull/888	H-02	Adds governance function to manually push the era forward
https://github.com/reserve-protocol/protocol/pull/876	M-01	Allow settle trade when paused or frozen
https://github.com/reserve-protocol/protocol/pull/873 & https://github.com/reserve-protocol/protocol/pull/869	M-02	Disable dutch auctions on a per-collateral basis, use 4-step dutch trade curve
https://github.com/reserve-protocol/protocol/pull/878	M-03	Distribute revenue in setDistribution
https://github.com/reserve-protocol/protocol/pull/885	M-04	Update payout variables if melt fails during setRatio
https://github.com/reserve-protocol/protocol-private/pull/15	M-06	Use lotPrice()
https://github.com/reserve-protocol/protocol-private/pull/7	M-07	Refresh before selling rewards, refactor revenue & distro
https://github.com/reserve-protocol/protocol/pull/857	M-08	payoutRewards before freeze and update payoutLastPaid before unfreeze
https://github.com/reserve-protocol/protocol-private/pull/3	M-09	Payout rewards during cancelUnstake
https://github.com/reserve-protocol/protocol/pull/886	M-10	Add oracle deprecation check

URL	Mitigation of	Purpose
https://github.com/reserve-protocol/protocol/pull/857	M-11	Change gas reservation policy in AssetRegistry

Out of Scope

- M-05: Lack of claimRewards when manageToken in RevenueTrader
- M-12: Custom redemption can be used to get more than RToken value, when an upwards depeg occurs

Mitigation Review Summary

Original Issue	Status	Full Details
H-01	Mitigation Confirmed	Reports from ronnyx2017, 0xA5DF, and rvierdiiev
H-02	Mitigation Confirmed	Reports from 0xA5DF, ronnyx2017, and rvierdiiev
M-01	Mitigation Confirmed	Reports from rvierdiiev, 0xA5DF, and ronnyx2017
M-02	Mitigation Errors	Reports from ronnyx2017 and 0xA5DF - details also shared below
M-03	Mitigation Error	Reports from 0xA5DF and rvierdiiev - details also shared below
M-04	Mitigation Error	Report from 0xA5DF - details also shared below
M-05	Sponsor Disputed	-
M-06	Mitigation Confirmed	Reports from rvierdiiev, 0xA5DF, and ronnyx2017
M-07	Mitigation Confirmed	Reports from ronnyx2017, 0xA5DF, and rvierdiiev
M-08	Mitigation Confirmed	Reports from rvierdiiev, 0xA5DF, and ronnyx2017
M-09	Mitigation Confirmed	Reports from rvierdiiev, 0xA5DF, and ronnyx2017
M-10	Mitigation Confirmed	Reports from 0xA5DF, ronnyx2017, and rvierdiiev

Original Issue	Status	Full Details
M-11	Mitigation Confirmed	Reports from ronnyx2017, 0xA5DF, and rvierdiiev
M-12	Sponsor Acknowledged	-

During their review, the wardens surfaced several mitigation errors. These consisted of 4 Medium severity issues. See below for details.

M-02 Mitigation Error: `dutchTradeDisabled[erc20]` gives governance an incentive to disable RSR auctions

Submitted by ronnyx2017

Severity: Medium

Lines of Code: Broker.sol#L213-L216

The mitigation adds different disable flags for GnosisTrade and DutchTrade. It can disable dutch trades by specific collateral. But it has serious problem with overall economic model design.

The traders Broker contract are under control of the governance. The governance proposals are voted by stRSR stakers. And if the RToken is undercollateralized, the staking RSR will be sold for collaterals. In order to prevent this from happening, the governance(stakers) have every incentive to block the rsr auction. Although governance also can set disable flag for trade broker in the previous version of mitigation, there is a difference made it impossible to do so in previous versions.

In the pre version, there is only one disable flag that disables any trades for any token. So if the governance votes for disable trades, the RToken users will find that they can't derive any gain from RToken. So no one would try to issue RToken by their collateral. It is also unacceptable for governance.

But after the mitigation, the governance can decide only disable the DutchTrade for RSR. And they can initiate a proposal about enable RSR trade -> open openTrade -> re-disable RSR trade to ensure their own gains. And most importantly, this behavior seems to do no harm to RToken holders just on the face of it, and it therefore does not threaten RToken issuance.

So in order to prevent the undercollateralized case, dutchTradeDisabled[erc20] gives governance every incentive to disable RSR auctions.

Impact

When RToken is undercollateralized, disabling RSR trade will force users into redeeming from RToken baskets. It will lead to even greater depeg, and the RToken users will bear all the losses, but the RSR stakers can emerge unscathed.

Proof of Concept

StRSR stakers can initiate such a proposal to prevent staking RSR auctions:

1. Call `Broker.setBatchTradeDisabled(bool disabled)` to disable any GnosisTrade.
2. And call `setDutchTradeDisabled(RSR_address, true)` to disable RSR DutchTrade.

Recommended Mitigation Steps

The `dutchTradeDisabled` flag of RSR should not be set to true directly by governance in the `Broker.setDutchTradeDisabled` function. Add a check like that:

```
    require(!(disabled && rsrTrader.tokenToBuy()==erc20),"xxxxxxx");
```

Assessed type

Context

tbrent (Reserve) confirmed and commented:

> Anticipating restricting governance to only be able to *enable* batch trade, or dutch trade.

pmckelvy1 (Reserve) commented:

> Implemented here and here.

M-02 Mitigation Error: Attacker might disable trading by faking a report violation

Submitted by 0xA5DF

Severity: Medium

Lines of Code: DutchTrade.sol#L212-L214

Dutch trade now creates a report violation whenever the price is x1.5 then the best price.

The issue is that the attacker can fake a report violation by buying with the higher price. Since revenue traders don't have a minimum trade amount that can cost the attacker near zero funds.

Mitigation might be to create violation report only if the price is high and the total value of the sell is above some threshold.

tbrent (Reserve) confirmed

pmckelvy1 (Reserve) commented:

> Fixed here - only rebalancing trades can disable dutch trades in this manner.

M-03 Mitigation Error: Funds aren't distributed before changing distribution

Submitted by 0xA5DF*, also found by* rvierdiiev

Severity: Medium

Lines of Code: Distributor.sol#L59-L63

Mitigation does solve the issue; however there's a wider issue here that funds aren't distributed before set distribution is executed.

Fully mitigating the issue might not be possible, as it'd require to send from the backing manager to revenue trader and sell all assets for the `tokenToBuy`. But we can at least distribute the current balance before changing the distribution.

tbrent (Reserve) confirmed and commented:

> Anticipating adding a try-catch at the start of `setDistribution()` targeting `RevenueTrader.distributeTokenToBuy()`

pmckelvy1 (Reserve) commented:

> Anticipating adding a try-catch at the start of `setDistribution()` targeting `RevenueTrader.distributeTokenToBuy()`

> Added here.

M-04 Mitigation Error: Furnace would melt less than intended

Submitted by 0xA5DF

Severity: Medium

Lines of Code: Furnace.sol#L92-L105

We traded one problem with another here. The original issue was that in case `melt()` fails then the distribution would use the new rate for previous periods as well.

The issue now is that in case of a failure (e.g. paused or frozen) we simply don't melt for the previous period. Meaning RToken holders would get deprived of the melting they're supposed to get.

This is especially noticeable when the ratio has been decreased and the balance didn't grow much, in that case we do more harm than good by updating `lastPayout` and `lastPayoutBal`.

A better mitigation might be to update the `lastPayout` in a way that would reflect the melting that should be distributed.

tbrent (Reserve) acknowledged and commented:

> I think this can only happen when frozen, not while paused. `Furnace.melt()` and `RToken.melt()` succeed while paused.

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 solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.