SOLIDIFIED

Audit Report for Reserve Protocol - October 31, 2022

Summary

Audit Report prepared by Solidified covering the Reserve Protocol.

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code below. The final debrief took place on September 12, 2022, and the results are presented here.

Audited Files

The source code has been supplied in the following source code repository:

Repo: https://github.com/reserve-protocol/protocol
Commit hash: `946d9b101dd77275c6cbfe0bfe9457927bd221a9`

```
contracts
├── facade
│   ├── Facade.sol
│   ├── FacadeWrite.sol
│   └── lib
│       └── FacadeWriteLib.sol
├── interfaces
│   ├── IAsset.sol
│   ├── IAssetRegistry.sol
│   ├── IBackingManager.sol
│   ├── IBasketHandler.sol
│   ├── IBroker.sol
│   ├── IComponent.sol
│   ├── IDeployer.sol
│   ├── IDistributor.sol
│   ├── IFacade.sol
│   ├── IFacadeWrite.sol
│   ├── IFurnace.sol
│   ├── IGnosis.sol
│   ├── IMain.sol
│   ├── IRToken.sol
│   ├── IRevenueTrader.sol
```

```
│   ├── IRewardable.sol
│   ├── IStRSR.sol
│   ├── IStRSRVotes.sol
│   ├── ITrade.sol
│   └── ITrading.sol
├── libraries
│   ├── Fixed.sol
│   ├── RedemptionBattery.sol
│   └── test
│       └── FixedCallerMock.sol
├── mixins
│   ├── Auth.sol
│   └── ComponentRegistry.sol
├── p0
│   ├── AssetRegistry.sol
│   ├── BackingManager.sol
│   ├── BasketHandler.sol
│   ├── Broker.sol
│   ├── Deployer.sol
│   ├── Distributor.sol
│   ├── Furnace.sol
│   ├── Main.sol
│   ├── RToken.sol
│   ├── RevenueTrader.sol
│   ├── StRSR.sol
│   └── mixins
│       ├── Component.sol
│       ├── Rewardable.sol
│       ├── Trading.sol
│       └── TradingLib.sol
├── p1
│   ├── AssetRegistry.sol
│   ├── BackingManager.sol
│   ├── BasketHandler.sol
│   ├── Broker.sol
│   ├── Deployer.sol
│   ├── Distributor.sol
│   ├── Furnace.sol
│   ├── Main.sol
│   ├── RToken.sol
│   ├── RevenueTrader.sol
```

```
│   ├── StRSR.sol
│   ├── StRSRVotes.sol
│   └── mixins
│       ├── Component.sol
│       ├── RewardableLib.sol
│       ├── Trading.sol
│       └── TradingLib.sol
└── plugins
    ├── aave
    │   ├── ERC20.sol
    │   ├── IAToken.sol
    │   ├── IAaveIncentivesController.sol
    │   ├── IStaticATokenLM.sol
    │   ├── RayMathNoRounding.sol
    │   ├── StaticATokenErrors.sol
    │   └── StaticATokenLM.sol
    ├── assets
    │   ├── ATokenFiatCollateral.sol
    │   ├── AbstractCollateral.sol
    │   ├── Asset.sol
    │   ├── CTokenFiatCollateral.sol
    │   ├── CTokenNonFiatCollateral.sol
    │   ├── CTokenSelfReferentialCollateral.sol
    │   ├── EURFiatCollateral.sol
    │   ├── FiatCollateral.sol
    │   ├── NonFiatCollateral.sol
    │   ├── OracleLib.sol
    │   ├── RTokenAsset.sol
    │   ├── RTokenPricingLib.sol
    │   └── SelfReferentialCollateral.sol
    ├── governance
    │   └── Governance.sol
    ├── mocks
    │   ├── ATokenMock.sol
    │   ├── AaveLendingPoolMock.sol
    │   ├── BadCollateralPlugin.sol
    │   ├── BadERC20.sol
    │   ├── CTokenMock.sol
    │   ├── ChainlinkMock.sol
    │   ├── ComptrollerMock.sol
    │   ├── ERC20Mock.sol
```

```
│   ├── EasyAuction.sol
│   ├── GnosisMock.sol
│   ├── InvalidATokenFiatCollateralMock.sol
│   ├── InvalidBrokerMock.sol
│   ├── NontrivialPegCollateral.sol
│   ├── RTokenCollateral.sol
│   ├── SelfdestructTransferMock.sol
│   ├── USDCMock.sol
│   ├── WETH.sol
│   ├── upgrades
│   │   ├── AssetRegistryV2.sol
│   │   ├── BackingManagerV2.sol
│   │   ├── BasketHandlerV2.sol
│   │   ├── BrokerV2.sol
│   │   ├── DistributorV2.sol
│   │   ├── FurnaceV2.sol
│   │   ├── MainV2.sol
│   │   ├── RTokenV2.sol
│   │   ├── RevenueTraderV2.sol
│   │   └── StRSRV2.sol
│   └── vendored
│       └── EasyAuction.sol
└── trading
    └── GnosisTrade.sol
```

Intended Behavior

The audited codebase implements a generic framework to issue tokens that are backed by a rebalancing basket of collateral.

Findings

Smart contract audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of a smart contract system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**.

Note, that high complexity or lower test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than a security audit and vice versa.

Criteria	Status	Comment
Code complexity	High	-
Code readability and clarity	High	-
Level of Documentation	Medium	-
Test Coverage	High	-

Issues Found

Solidified found that the Reserve Protocol contracts contain 3 critical issues, 3 major issues, 9 minor issues, and 7 informational notes.

We recommend issues are amended, while informational notes are up to the team's discretion, as they refer to best practices.

Issue #	Description	Severity	Status
1	RToken.sol (P1): Attacker can cancel issuance multiple times and steal tokens of other users	Critical	Fixed
2	BasketHandler.sol (P1): Wrong behavior when erc20s contains same token multiple times	Critical	Fixed
3	EURFiatCollateral.sol: refresh() may revert	Critical	Fixed
4	RToken.sol (P0): Queue not properly cleaned up when issuing instantly	Major	Fixed
5	RToken.sol (P0): Reentrancy with ERC777 tokens	Major	Acknowledged
6	Market capitalization risk of the RSR token: suboptimal collateral properties introduce the risk of liquidity spirals	Major	Acknowledged
7	BasketHandler.sol: Basket can contain ERC20 token multiple times	Minor	Fixed
8	FacadeWrite.sol: Backup tokens allowance for RToken not set	Minor	Fixed
9	Fee-on-transfer tokens lead to problems	Minor	Acknowledged
10	RToken.sol: Unnecessary zero value transfers in redeem()	Minor	Fixed
11	StRSR.sol: permit() not compatible with smart contracts	Minor	Fixed

12	NonFiatCollateral.sol: maxTradeSize() may return wrong values because of an overflow	Minor	Fixed
13	Overly flexible governance design bears risks for long term stability	Minor	Acknowledged
14	Defaults of poorly designed RTokens will have implications on the liquidity of RSR as a collateral and thereby affect other RTokens	Minor	Acknowledged
15	BasketHandler.sol: add method allows duplicates	Minor	Fixed
16	Facade.sol: endIdForVest sometimes returns wrong values	Minor	Fixed
17	Facade.sol: getActCallData performs calls before returning calldata	Minor	Acknowledged
18	RedemptionBattery.sol: BLOCKS_PER_HOUR may be wrong	Note	Fixed
19	BasketHandler.sol: nonce is increased two times in copy	Note	Fixed
20	RToken.sol (P1): Invariant in comment for refundSpan does not hold	Note	Fixed
21	RToken.sol (P1): Gas optimizations for issue	Note	Fixed
22	RToken.sol (P1): Reentrancy attack possible if token is ever upgraded to ERC777 token	Note	Fixed
23	Asset.sol: Trading range parameter validation	Note	Fixed
24	BasketHandler.sol: Return value and the description on comment are different	Note	Fixed
25	String.sol: toLower does not handle all UTF-8 characters correctly	Note	Fixed

Critical Issues

1. RToken.sol (P1): Attacker can cancel issuance multiple times and steal tokens of other users

---

When `cancel()` is used to cancel from the right side (i.e. with `earliest` set to `false`), `queue.right` is set to the user-provided `endId`, but the refunded queue items with a higher id are not deleted from the queue. In `issue()`, `queue.right` is then increased, but the new issuance is appended to the queue (i.e., it is not at the new value of `queue.right`). An attacker can exploit this to steal tokens of other users like this:

- Cancel issuances from the right side
- Start a new issuance (with an arbitrarily small amount) to increase `queue.right`
- Cancel issuances from the left side

Proof Of Concept

The issue is illustrated in the following proof of concept:

```
  it('Exploit cancel', async function () {
    const issueAmount: BigNumber = bn('25000e18')
    const balBefore = await token0.balanceOf(addr1.address);

    // Provide approvals
    await token0.connect(addr1).approve(rToken.address, initialBal)
    await token1.connect(addr1).approve(rToken.address, initialBal)
    await token2.connect(addr1).approve(rToken.address, initialBal)
    await token3.connect(addr1).approve(rToken.address, initialBal)
    await token0.connect(addr2).approve(rToken.address, initialBal)
    await token1.connect(addr2).approve(rToken.address, initialBal)
    await token2.connect(addr2).approve(rToken.address, initialBal)
    await token3.connect(addr2).approve(rToken.address, initialBal)

    // Issue rTokens
    await rToken.connect(addr1).issue(issueAmount)
    await rToken.connect(addr1).issue(issueAmount.mul(2))
    await rToken.connect(addr1).issue(issueAmount)
```

```
    await rToken.connect(addr2).issue(issueAmount.mul(3)) // Part of this will be
stolen by addr1

    // Cancel with issuer
    await expect(rToken.connect(addr1).cancel(1, false))
      .to.emit(rToken, 'IssuancesCanceled')
      .withArgs(addr1.address, 1, 3, issueAmount.mul(3))

    await rToken.connect(addr1).issue(1) // To manipulate queue.right
    await expect(rToken.connect(addr1).cancel(2, true))
      .to.emit(rToken, 'IssuancesCanceled')
      .withArgs(addr1.address, 0, 2, issueAmount.mul(3))

    const balAfter = await token0.balanceOf(addr1.address)
    console.log(balBefore)
    console.log(balAfter)
  })
```

In this example, the attacker steals the tokens of `addr2`.

Recommendation
Delete the canceled queue items when canceling from the right side.

2. BasketHandler.sol (P1): Wrong behavior when erc20s contains same token multiple times

---

In the P1 implementation, when `handoutExcessAssets` is called with `duplicates` in `erc20s`, these assets will be handed out multiple times (because the calculation, which is performed first, is based on the balance), leading to an accidental undercollaterization.

Recommendation
Check that there are no duplicates in `erc20s`. Note that there are some tokens that have two addresses. If they are used as an underlying, it would not be sufficient to only check that there are no duplicates, as it would be possible to provide both addresses in `erc20s`.

3. EURFiatCollateral.sol: refresh() may revert

The `refresh()` function of an asset should never revert, as this completely disables an RToken (because every call to `refresh()` of the asset registry then reverts). Instead, `priceable` should be set to `false`. However, the `refresh()` of `EURFiatCollateral` may revert. When `p2 == 0`, there will be a division by 0 because of `p1.div(p2)`.

Recommendation

Only perform the division if `p2 > 0`.

Major Issues

4. RToken.sol (P0): Queue not properly cleaned up when issuing instantly

When a issuance is completed instantly in the RToken of P0, `delete issuances[issuer][index]` is used to remove the issuance from the queue. However, this does not actually remove the item from the array, it only resets all struct values to their default values. This can be very problematic in the following scenario (demonstrated in the proof of concept below):

- A issuance that cannot be completed instantly is added to the queue.
- Later on, the same user adds an issuance that can be completed instantly.
- Now, `issuances` contains an entry with `processed = false` and `basketNonce = 0` (the one that was completed instantly and was not properly removed). When the user tries to vest the first issuance, this does not succeed, because `refundAndClearStaleIssuances` is first called, which refunds all issuances (because of this one entry).

Proof Of Concept

The issue is illustrated in the following proof of concept:

```
  it('Instant Issuance PoC', async function () {
```

```
    const instantIssue: BigNumber = MIN_ISSUANCE_PER_BLOCK.sub(1)
    const issueAmount: BigNumber = MIN_ISSUANCE_PER_BLOCK.mul(3)

    // Provide approvals
    await token0.connect(addr1).approve(rToken.address, initialBal)
    await token1.connect(addr1).approve(rToken.address, initialBal)
    await token2.connect(addr1).approve(rToken.address, initialBal)
    await token3.connect(addr1).approve(rToken.address, initialBal)

    // Issue rTokens
    await rToken.connect(addr1).issue(issueAmount)

    // Attempt to vest
    await expect(rToken.vest(addr1.address, 1)).to.be.revertedWith('not ready')

    await advanceBlocks(5)
    // Issue rTokens
    await expect(rToken.connect(addr1).issue(instantIssue)).to.emit(rToken,
'IssuanceStarted')

    // Should vest now
    await expect(await rToken.vest(addr1.address, 1)).to.emit(rToken,
'IssuancesCompleted')
  })
```

Running the test with P1 succeeds (as it should), whereas it fails with P0.

Recommendation

Delete the item from `issuances` with `pop()`.

5. `RToken.sol` (P0): Reentrancy with ERC777 tokens

When canceling or refunding in the `RToken` of P1, `iss.processed` is set to `true` after the token transfer. This is problematic when ERC777 tokens are used. In such cases, the receiver can

reenter the function (in the `tokensReceived` hook) and cancel or request a refund again. This could be used to completely drain the contract.

Recommendation

Set `iss.processed` to `true` before the transfer.

Status

Acknowledged: “This is true, but not particularly relevant, as P0 is for testing and will not be deployed on mainnet.”

6. Market capitalization risk of the RSR token: suboptimal collateral properties introduce the risk of liquidity spirals

---

The RSR token can be used to collateralize/insure any RToken. At the same time, and just because of its so far limited market capitalization and liquidity when compared to other on-chain assets (such as Ether or Bitcoin) it is not the perfect collateral. This may lead to suboptimal staking incentives, when the price of RSR fluctuates strongly. A swift change in the RSR price may incentivize stakers to unlock their tokens and sell them at market conditions. This is true for both strong upwards and downward movements of the price. While there are little systemic risk implications, in upward trending markets (likely people will sell and new buyers will eventually restake) in markets with strong downwards dynamics liquidity spirals may occur.
The intuition behind these spirals is as follows: downward trending RSR prices will increase the incentive for stakers to unstake while at the same time the collateral value of RSR decreases making it more likely that a larger amount might be seized in case of a default event - which is again more likely during doward trending markets. As highlighted in Brunnermeier and Pedersen (2009) such market conditions can reinforce themselves and are particularly severe when the liquidity of the collateral is low. In an adverse scenario, where one of the RTokens defaults this could lead to so strong selling pressure of the RSR token that all RTokens are undercapitalized from an insurance perspective - i.e. that the market capitalization of the staked RSR tokens is significantly lower than the likelihood-of-default adjusted capitalization-need to provide insurance. In such a scenario, additional selling pressure will be induced by shrinking market capitalizations of RTokens as the holders will likely prefer to liquidate their RToken for

the underlying stablecoin basket. Such a shrinking market capitalization might again disincentivize staking.

In the real world mostly government bonds or some sort of real estate backed security are considered as the most liquid collateral. While these are not yet available on chain, RSR will likely not be the most liquid collateral available to collateralize a unified stable currency at its inception.

Recommendation

We recommend either replacing RSR as a collateral asset with a more liquid collateral asset with more market capitalization, or ideally augmenting it with a combination of such assets, which are characterized by low volatility, high market capitalization and high liquidity.

Status

Acknowledged:
While this is an edge case that is a true positive, we don't believe it is substantially relevant to be a "major" issue. This is mostly because we have not designed the protocol to use RSR or staked RSR for basket collateral of an RToken. Therefore, any price drops in RSR would only affect the collateral in the insurance pool.

However, it is technically possible to use RSR or stRSR as collateral for other RTokens, we just expect those to be much smaller than typical RTokens backed by fiat-backed stablecoins or other liquid collateral types.

In the case of a large RToken's non-RSR collateral defaulting and the protocol needing to liquidate a large block of insurance pool RSR, depending on RSR liquidity at the time, this may cause a price drop which would lead to other insurance pools shrinking in comparison to their respective RToken market caps. However, since these are still collateralized with other assets, this won't affect the redeem-ability or peg of the other RTokens (unless they use the same non-RSR defaulting collateral).

Liquidity for RSR will be important. After we launch, we expect there to be more on chain liquidity with uniswap/curve pools and we have an Aave proposal approved to be a borrowable

asset. We're expecting that the onchain and offchain liquidity will only improve as there are more uses for the governance token beyond speculation as is the current situation.

That all being said, this is an edge case weakness known by the team and we have been exploring ways to introduce other forms of collateral in the insurance pools and will continue to research possibilties to evolve the soundness of the insurance system over time.

Minor Issues

7. BasketHandler.sol: Basket can contain ERC20 token multiple times

---

BasketHandler does not validate that the ERC20 tokens are unique and `config.erc20s` can contain the same token multiple times, which leads to unexpected behavior: While `config.erc20s` will contain the token multiple times, it will only be once in `config.targetAmts` (with the amount of the latest item). This causes a problem in `_switchBaskets`, where this amount is then used multiple times.

Recommendation

Either do not allow the same ERC20 token multiple times or handle it correctly (such that the individual amounts are added up, instead of multiplying the amount of the last one).

8. FacadeWrite.sol: Backup tokens allowance for RToken not set

---

In contrast to the tokens in the primary basket, `grantRTokenAllowance` is not called for the backup tokens. Therefore, redemption will fail after a failover. While the error is recoverable (as the function `grantRTokenAllowance` is callable by anyone), it requires manual intervention.

Recommendation

Call `grantRTokenAllowance` also for the backup tokens.

9. Fee-on-transfer tokens lead to problems

In multiple places (e.g., `RToken.issue`), it is assumed that the amount that is passed to `transfer` / `transferFrom` will also arrive at the recipient. This is not true for tokens that charge a fee on transfer. There, the amount that for instance is transferred to the `BasketHandler` will not be equal to the amount that was returned by `quote`.

Recommendation
If fee-on-transfer tokens are not supported in the underlying baskets, this should be documented. Otherwise, the system would need to incorporate fees into the calculations.

Status
Acknowledged: “We agree that fee-on-transfer tokens lead to problems, and consider this a design flaw of fee-on-transfer tokens. If someone wants to include a fee-on-transfer token as an RToken asset, creating a simple wrapper that circumvents the fee is little more effort than is already needed to write the token’s Asset plugin. We will document this distinction.”

10. RToken.sol: Unnecessary zero value transfers in redeem()

In `redeem()` (P0 & P1), a transfer is initiated even if `amounts[i]` is 0. This is a waste of gas and even reverts for some tokens (e.g., LEND), although this is rare, as it is a violation of EIP-20.

Recommendation
Only initiate a transfer if `amounts[i]` is greater than 0.

11. StRSR.sol: permit() not compatible with smart contracts

`permit()` in `StRSR` uses `ECDSA.recover` to handle signatures. Unlike `SignatureChecker.isValidSignatureNow`, this is not compatible with EIP1271, meaning that smart contracts (and especially smart contract wallets) will not be able to approve with signatures.

Recommendation

Consider using `SignatureChecker.isValidSignatureNow`, which first tries to do ECDSA recovery and then follows EIP1271.

12. NonFiatCollateral.sol: maxTradeSize() may return wrong values because of an overflow

---

In the calculation `p = uint192((uint256(p) * p2) / FIX_ONE)`, an overflow may occur, leading to wrong values for p.

Recommendation

Consider using `mul` like in `price()`. Then, the calculation would revert in such a scenario and the system would not continue calculating with wrong values.

13. Overly flexible governance design bears risks for long term stability

---

The protocol has a well spelled out strategic vision to create a universal stable coin featuring century-to-century stability. This strategy needs to be reflected in the design of the governance mechanism and restrict the possibility of governance failures and attacks.
All major central banks have an operational strategy which is overhauled only every couple of years. This makes the functioning day-to-day reliable and the strategy testable and improvable in a scientific sense and from a year-to-year perspective. This also a well established result that is relied on in economic academia and applied economics `Kydland and Prescott (1977)` show that strategic rules are - form an optimal control theory perspective - superior to discrete decision making because they tend to reduce the time-inconsistency problem in dynamic stochastic systems such as monetary system.

This would suggest a more rigid and less open governance system that basically negotiates the weighting function of the stable coins instead of the individual weights.
Most stock market indices, economic benchmarks, and headline figures are composed of weighted elements that are a function of market capitalization (MSCI World), a basket of goods (to compute inflation & payment measures) or production/trade (`special drawing rights`).
In addition, `modern portfolio theory` highlights the importance of including the correlation structure of risky assets, when constructing a portfolio.

As governance participation in new protocols has been turned out low in the past and typically direct governance makes it easy for whales or influencers to propose and push through a decision. Governance around a weighting function would make it significantly harder to push through a malicious proposal. In addition, it would reduce the long run risk of uninformed decision making that is not aligned with the overall vision of the protocol. Thereby it would reduce tail-risks (such that materialize with low probability, high impact) and protect investors.

Recommendation
We recommend to limit the powers of governance significantly. Governance should not be able to alter individual weights of RTokens but only weighting functions. Governance should be limited to small and strategic changes with a governance cycle that is independent of the day-to-day business. We do not have a preference for a specific cycle, but flag that special drawing rights weights are only updated every 5 years, whereas the weighting function methodology is even updated more seldomly.

Status
Acknowledged: “It is flexible on purpose in the beginning because we agree that current token voting mechanisms in the marketplace are insufficient for the 100 year currency management problem. Flexible governance design leaves room for actual governance design. So, we have satisfactory immediate governance, plans to improve in the short-term, and substantial research efforts underway to improve in the longer term (2-3 years).”

14. Defaults of poorly designed RTokens will have implications on the liquidity of RSR as a collateral and thereby affect other RTokens

---

RSR can be used to collateralize any RToken, this creates the following incentives for creators or significant holders of poorly designed stablecoins - particularly those with a ponzi-like model of prolonged unsustainable high yields.

1. Vote the own token into any existing RToken, to benefit from its backing of RSR in case of a default.
2. Create an own high-yield RToken and attract RSR holders to stake.

This will either increase the risk of defaults of existing RTokens or in the second case lead to risky RTokens that are prone to default.

Making the creation of any RToken available to the general public might damage the brand value of the RSR token, decrease its liquidity and increase its volatility. This has an indirect effect on other RTokens, which are backed by the RSR token.

Recommendation

We recommend creating new RSR tokens for each of the RTokens, unless RSR governance decides to whitelist a newly created RToken.

Status

Acknowledged: “Yes. We agree that large RTokens that have a high percent of the total market cap of RTokens generally held in risky collateral could lead to poor incentives that would endanger the larger system.

This is something we will be working to avoid. We believe that sufficient PMF exists for safer RTokens and will be working within the governance forums of various RToken forums to ensure that everyone is aware of the dangers that might come about from a larger, risky RToken.

We prefer the tradeoffs of our existing design to the recommendation, as the recommendation trades off the weighty benefits of a shared governance token.”

15. BasketHandler.sol: add method allows duplicates

The method add in BasketLib will add duplicate addresses to the array if the weight of the already added token is zero.

```
78| self.erc20s.push(tok);
```

Recommendation

Consider validating whether the weight of a token is greater than zero.

16. Facade.sol: endIdForVest sometimes returns wrong values

`Facade.endIdForVest` uses a `<=` when checking the end of the queue, whereas `<` is used within the binary search. This leads to the following inconsistency: Imagine the queue is initially [10, 11, 15, 20] and the current block number is 15. A call to endIdForVest would return

`queueRight` (i.e., the item with block number 20) because `15 <= 15`. If the queue would be [10, 11, 15, 20, 20] and the current block number is still 15, `right` within the binary search would be set to the index of the item with block number 15.

Recommendation
Use `<=` within the binary search which would also be in line with the comment (`queue[left].when <= block.timestamp`)

17. Facade.sol: getActCallData performs calls before returning calldata

---

Before returning the calldata, `Facade.getActCallData` calls `furnace.melt` or `stRSR.payoutRewards`. Therefore, melting or reward payout opportunities may already be eliminated in the function itself and not by the MEV searcher.

Recommendation
Only return the calldata in the function (as for the other parts of the system that are checked there).

Status
Acknowledged: “the function `getActCallData()` is intended to be executed only via a `callStatic` method - which is wildly cheaper, and so is reasonably what you’d expect an MEV searcher to do.”

Informational Notes

18. RedemptionBattery.sol: BLOCKS_PER_HOUR may be wrong

`BLOCKS_PER_HOUR` is set to 300 in `RedemptionBattery`. Before the merge, this value is wrong (and differs every hour). But even after the merge, this is an upper bound and there are only 300 blocks per hour if all validators submit a block in their slot (see https://0xfoobar.substack.com/p/ethereum-proof-of-stake for details).

Recommendation

Consider working with timestamps instead of blocks to get more precise values for `charge`.

19. BasketHandler.sol: nonce is increased two times in copy

The function `empty` that is called within `copy` already increases the nonce by one, increasing it another time is unnecessary.

Recommendation
Consider removing the second increment.

20. RToken.sol (P1): Invariant in comment for refundSpan does not hold

Before `refundSpan`, it is mentioned that `/// after: queue.left == queue.right`. However, that does not ever hold. For instance, when refunding from the end of the queue, `queue.left < queue.right` always holds.

Recommendation
Update the specification / documentation.

21. RToken.sol (P1): Gas optimizations for issue

- Unnecessary refund for the first call: When a user calls `issue` for the first time, the `basketNonce` of his queue is 0, meaning that `queue.basketNonce != basketNonce` is `true` and the logic for refunding is executed unnecessarily.
- Collateral status checked two times: When it is checked if the queue can be bypassed, it is checked if the collateral is sound. However, there is already a `require` statement in the function that checks that. This code will therefore be never executed when the collateral is unsound.

Recommendation

- Check if `queue.basketNonce != 0` and do not initiate a refund in such cases to save gas.
- Remove the second check for the status.

22. RToken.sol (P1): Reentrancy attack possible if token is ever upgraded to ERC777 token

In `issue` and `vestUpTo`, `_mint` is called before `basketsNeeded` is updated. This is no problem in the current implementation (where `RToken` is an ERC20 token), but if the system were ever to be updated to use the ERC777 standard, it would become problematic. Then, a user could reenter and `totalSupply()` would be updated, whereas `basketsNeeded` would still hold the old value.

Recommendation

If you do not plan to update to the ERC777 standard, nothing must be changed. But you could still reorder the statements such that this attack never becomes possible.

23. Asset.sol: Trading range parameter validation

While it is validated that `tradingRange_.minAmt > 0` and `tradingRange_.maxAmt > 0`, it is currently not validated that the maxAmt is greater than the minAmt. A situation where this is not the case would lead to wrong values for `minTradeSize()` / `maxTradeSize()`.

Recommendation

Consider validating that `maxAmt` is greater than `minAmt`.

24. BasketHandler.sol: Return value and the description on comment are different

The method `basketsHeldby` returns `FIX_ZERO` for an empty basket whereas the comment above describes it should return `FIX_MAX`.

Recommendation
Consider fixing the comment to reflect the actual return value.

25. String.sol: toLower does not handle all UTF-8 characters correctly

A comment for `toLower` states that “This turns out to work for UTF-8 in general.”. However, this is not true in practice. For instance, the letter Ö has multiple Unicode representations. It can be represented as 0x4FCC88 (LATIN CAPITAL LETTER O and COMBINING DIARESIS), but it also can be represented as 0xC396 (LATIN CAPITAL LETTER O WITH DIARESIS). The first representation will be correctly handled by `toLower`, whereas it will not change the second representation.

Recommendation
Because the function is currently only used on the symbol, this should not be a problem in practice. Consider clarifying the comment such that no wrong usages are introduced in the future.

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of Reserve Protocol or its products. This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors of Solidified platform from legal and financial liability.

Oak Security GmbH