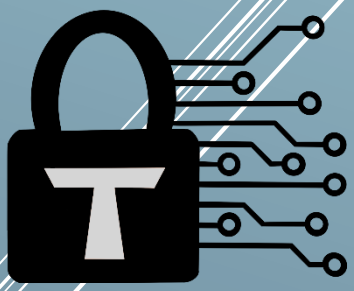
Trust Security

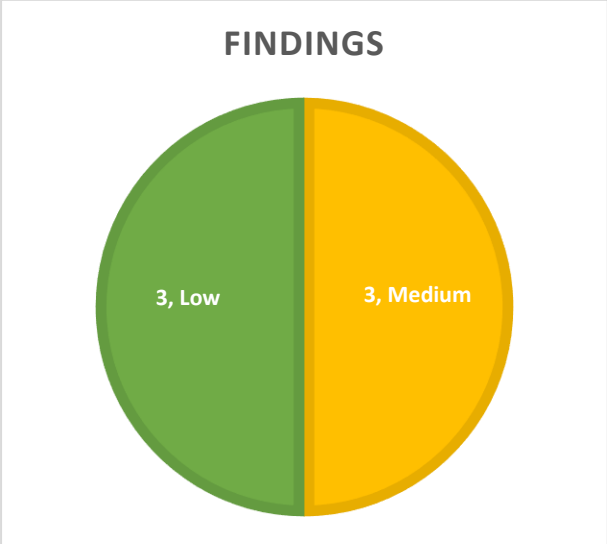


Smart Contract Audit

Reserve Protocol – Release 4.0.0

26/07/24

Executive summary

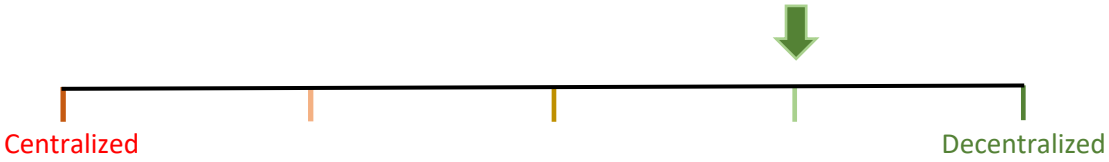


Category	Stablecoin
Auditor	HollaDieWaldfee cccz

Findings

Severity	Total	Fixed	Acknowledged
High	-	-	-
Medium	3	3	-
Low	3	1	2

Centralization score



Signature

Digitally signed by Trust
Date: 2024.07.26 12:10:17 +03'00'

Document properties

Versioning

Version	Date	Description
0.1	26/07/2024	Client report

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security serves as a long-term security partner of the Reserve Protocol. It has conducted the audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Additional recommendations have been given when appropriate.

Scope

The following PRs are in scope of the audit:

- Release 4.0.0

Repository details

- **Repository URL:** https://github.com/reserve-protocol/protocol
- **Commit hash:** 93d2831b2c5885ad69a27403de7436f7c4ca04b8
- **Mitigation hash:** 5a82ec4591b42761b6c5603c8592d7e2bfa1b82c

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

About the Auditors

HollaDieWaldfee is a renowned security expert with a track record of multiple first places in competitive audits. He is a Lead Senior Watson at Sherlock and Lead Auditor for Trust Security and Renascence Labs.

A top competitor in audit contests, cccz has achieved superstar status in the security space. He is a Black Hat / DEFCON speaker with rich experience in both traditional and blockchain security.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Findings

Medium severity findings

TRST-M-1 Gnosis auctions cannot settle when bought amount exceeds type(uint96).max

- **Category:** Overflow issues
- **Source:** GnosisTrade.sol
- **Status:** Fixed

Description

According to the Gnosis documentation, auctions for which the raised amount exceeds **type(uint96).max** (about **7.9e28**), cannot be settled.

This behavior has been confirmed in a test case. Due to the support for 21 decimal tokens with the requirement that one whole token is worth at least $1, the amount of funds needed to overflow the auction is $79 million. In fact, initially 27 decimal tokens were supposed to be supported, but these plans were quickly discarded upon discovering the vulnerability, so going forward the finding only deals with 21 decimal tokens.

The impact of overflowing a Gnosis auction is that the funds are locked and forever lost. It is not possible to recover them.

Recommended mitigation

Gnosis auctions are open to anyone that wants to place a bid. Therefore, it is necessary that the value of funds offered in an auction is sufficiently low such that:

1. Settlement at market price does not overflow **uint96**.
2. The gap between buy amount according to market price and the amount that overflows **uint96** is sufficiently large to prevent griefing attempts.

Both requirements lead to the conclusion that the market value of the sold tokens must not reach tens of millions of USD.

It has been determined that **maxTradeVolume**, which is set for each plugin, is insufficient to restrict the value of the sold tokens. This is since **maxSell** is calculated based on the **sellLow** price, which during the price decay phase, decays down to **0**. As such, the allowed trade volume effectively goes toward infinity during price decay.

It is suggested to use the **sellHigh** price to calculate **maxTradeVolume** which is reasonable to rely upon as an upper bound for the price of the sold tokens, even during decay.

Any trade that is opened must go through *TradeLib.prepareTradeSell()*, which then reliably limits the sold value at **maxTradeVolume**. The cost of griefing is thus **fundsToOverflow – maxTradeVolume** where **fundsToOverflow = $79 million**.

It must be considered that buy tokens might not be pegged to USD, in theory they can be pegged to any unit. This means they can decrease in terms of their USD valuation without

becoming **IFFY** or **DISABLED**. So, the $79 million number is not a lower bound for the funds it takes to cause an overflow in general.

To use the token price in such a context that, if the price is wrong, it can lead to a loss of funds, it must be documented that plugin prices must be manipulation resistant. Such a strong requirement has not existed so far.

An exception to using **sellHigh** for restricting the **maxTradeVolume** should be made when **sellHigh = FIX_MAX**. This is because **sellHigh = FIX_MAX** would mean that no funds can be sold, however this is not the intended behavior. *RecollateralizationLib* does not sell tokens for which **sellHigh** has decayed to **FIX_MAX**. Therefore, **sellHigh = FIX_MAX** is only relevant for *RevenueTrader* which generally does not deal with large amounts of funds, and which should just sell all of the sell tokens when **sellHigh = FIX_MAX**.

Team response

- Documented support for 21 decimal tokens and token value requirements.
- Documented requirement for manipulation resistant prices.
- Use **sellHigh** price instead of **sellLow** price for trade sizing.
- Revert when **maxSell <= 1** instead of selling the full sell amount **s**.
- Explicitly revert when **sellHigh = FIX_MAX && sellLow > 0** instead of relying on upstream assumptions in plugins.

Mitigation review

The recommended approach has been implemented. It relies on token valuation constraints that are documented in the repository, but not checked for explicitly.

It is worth highlighting that the protocol security is very sensitive to using tokens with a valuation or decimals outside of the expected ranges. RToken governance must ensure that all tokens that the protocol interacts with satisfy these requirements.

TRST-M-2 RSR stakers are vulnerable to peg price arbitrage

- **Category:** Logical issues
- **Source:** BasketHandler.sol
- **Status:** Fixed

Description

Each collateral plugin is configured with a **defaultThreshold**. It is a percentage which the observed **target / ref** rate (rate reported by the oracle) is allowed to be below or above the expected peg price (for example 1 for **BTC / WBTC**) before the collateral is marked **IFFY**.

RToken issuance does not get disabled if the deviation of the observed peg from the expected peg is less than **defaultThreshold**. For example, for **target = BTC**, **ref = WBTC** and **defaultThreshold = 2%**, issuance only gets paused when **BTC / WBTC < 0.98** or **BTC / WBTC > 1.02**. The amount of collateral needed to issue RToken is calculated with the assumption that **BTC / WBTC = 1**. When **BTC / WBTC** is less than the expected peg, a user can issue RTokens with less **target** than what the RToken should be backed by.

The problem is that the RSR overcollateralization ensures that in the long term the RToken is redeemable for certain target amounts. In the above example, when **BTC / WBTC = 0.98**, there are two possibilities:

1. **BTC / WBTC** does not decrease below 0.98, i.e. the user does not incur a loss in terms of the target unit BTC.
2. **BTC / WBTC** decreases below 0.98. A basket switch occurs and recollateralization buys a collateral that has not de-pegged, sourcing the missing funds from RSR stakers.

Both cases together illustrate that by issuing when **target / ref** drops below the expected peg, a user is protected from further downside while gaining the optionality that the price recovers.

Analysis shows that a targeted attack is unlikely to be profitable as it requires:

1. RToken parameters favorable to the attack (including big default thresholds, low diversification of collateral, collateral known to frequently de-peg).
2. External conditions that cause **target / ref** to move close to **pegBottom**.
3. Inactive / slow RToken Governance since Governance can freeze the attacker's funds.
4. Large capital requirement.

Due to the above conditions, it is more likely that regular users exploit the issue where RToken issuance is concentrated around periods of moderate de-pegs.

Still, the magnitude of the loss for RSR stakers is unbounded. The incentives for RToken issuance are aligned to completely wipe RSR stakers.

Recommended mitigation

The recommendation, that has been discussed in detail with the client, is to introduce an "issuance premium". If the observed peg price is below the expected peg price, token quantities for issuance are scaled such that actual target amounts paid to issue RToken match the target amounts in the basket. For example, if **BTC / WBTC = 0.98**, token quantities for issuance are scaled by **1 / 0.98 = ~1.02**.

This approach has different downsides, which is why the issuance premium is optional and can be set in *BasketHandler*.

1. There is a gap the size of one oracle error between the expected peg price and the market peg price when the issuance premium is guaranteed to be applied. For example, if the **BTC / WBTC** oracle error is 2%, the **BTC / WBTC** rate reported by the oracle can be **1**, while the market price has dropped to **0.98**. The issuance premium is only applied if the market price drops further. Having this error the size of one **oracleError** instead of **defaultThreshold** is a big improvement. In fact, it is a strict improvement since **oracleError < defaultThreshold**.
2. RToken issuers can be overcharged when the peg price reported by the oracle is lower than the market peg price. Issuance volume is therefore expected to be lower for RTokens that have the issuance premium enabled.

Team response

Fixed as described in the recommendation. The issuance premium is finalized at this commit.

Mitigation review

No issues have been found after reviewing the final commit besides the known downsides that are described above.

The issuance premium is calculated in *BasketHandler.issuancePremium()* and applied to issuances in *BasketHandler.quote()* conditioned on the **applyIssuancePremium** flag which is a governance parameter.

TRST-M-3 Target amount normalization is unsafe due to oracle errors

- **Category:** Logical issues
- **Source:** BasketHandler.sol
- **Status:** Fixed

Description

The mitigation for TRST-M-1 in the audit for release 3.2.0 has introduced target amount normalization for reweightable RTokens. Target amount normalization is needed to ensure the **UoA** value of the target amounts in the old prime basket matches the **UoA** value of the target amounts in the new prime basket. In other words, changing the prime basket must not lead to a loss for RSR stakers (increase in **UoA** value) or RToken holders (decrease in **UoA** value).

Target amount normalization scales the target amounts in the new prime basket to achieve matching **UoA** values.

```
// Scale targetAmts by the price ratio
newTargetAmts = new uint192[](len);
for (uint256 i = 0; i < len; ++i) {
    // {target/BU} = {target/BU} * {UoA/BU} / {UoA/BU}
    newTargetAmts[i] = targetAmts[i].mulDiv(price, newPrice, CEIL);
}
```

Reweightable RTokens are motivated by the intention to track an index where the components within the index can be changed.

In a use case where the components are changed frequently and the changes are small, the calculation errors can have unintended consequences. Generally, oracle errors are within **0.1%** to around **3%**. It is easy to see that a calculation error of this magnitude can change the target amount adjustment in an unintended way. Small adjustments can be lost in the noise of the oracle error. For example, due to an oracle error of **2%**, a target amount adjustment from **1 BTC** to **0.99 BTC** and **0.2 ETH** (assuming 0.01 BTC is worth 0.2 ETH) can end up as **1.02 * (0.99 BTC, 0.2 ETH) = (1.0098 BTC, 0.204 ETH)**.This has a completely unintended net effect of increasing the **UoA** value of baskets, even ending up with more BTC than in the beginning.

Frequent adjustments can drain RSR stakers if they need to compensate for a **UoA** value increase multiple times or cause a loss to RToken holders if the new target amounts tend to be set too low.

Consequently, target amount normalization is not safe to apply in certain (if not most) reweightable RToken use cases.

It must be noted that the above calculation illustrates a worst-case scenario to make a point that target amount normalization is not generally safe. The recommended mitigation is not a full fix. Instead, it removes the dangerous code, documents the requirement that RTokens need to implement their own safety checks, and allows the protocol to be extended by spells without upgrading the core contracts.

Recommended mitigation

It is recommended to remove target amount normalization from the core protocol. Instead, target amounts can be checked / adjusted in periphery contracts where the logic can be specific to the reweightable RToken's use case. Such periphery contracts can be implemented with the "spell" pattern that is currently used for upgrades.

The below diff removes target amount normalization for reweightable RTokens and instead exposes the *forceSetPrimeBaskset()* function that allows reweightable RTokens to set target amounts without any restrictions. By default, both non-reweightable and reweightable RTokens now require constant target amounts.

```
--- a/contracts/p1/BasketHandler.sol
+++ b/contracts/p1/BasketHandler.sol
@@ -193,21 +193,20 @@ contract BasketHandlerP1 is ComponentP1, IBasketHandler {
     /// @param targetAmts The target amounts (in) {target/BU} for the new prime
basket
     /// @custom:governance
     function setPrimeBasket(IERC20[] calldata erc20s, uint192[] calldata targetAmts)
external {
-        _setPrimeBasket(erc20s, targetAmts, true);
+        _setPrimeBasket(erc20s, targetAmts, false);
     }

-    /// Set the prime basket without reweighting targetAmts by UoA of the current
basket
+    /// Set the prime basket without requiring constant target amounts
     /// @param erc20s The collateral for the new prime basket
     /// @param targetAmts The target amounts (in) {target/BU} for the new prime
basket
     /// @custom:governance
     function forceSetPrimeBasket(IERC20[] calldata erc20s, uint192[] calldata
targetAmts) external {
-        _setPrimeBasket(erc20s, targetAmts, false);
+        _setPrimeBasket(erc20s, targetAmts, true);
     }

     /// Set the prime basket in the basket configuration, in terms of erc20s and
target amounts
     /// @param erc20s The collateral for the new prime basket
     /// @param targetAmts The target amounts (in) {target/BU} for the new prime
basket
-    /// @param normalize True iff targetAmts should be normalized by UoA to the
reference basket
     /// @custom:governance
     // checks:
     //   caller is OWNER
@@ -223,13 +222,13 @@ contract BasketHandlerP1 is ComponentP1, IBasketHandler {
     function _setPrimeBasket(
         IERC20[] calldata erc20s,
         uint192[] memory targetAmts,
-        bool normalize
+        bool disableTargetAmountCheck
     ) internal {
         requireGovernanceOnly();
         require(erc20s.length != 0 && erc20s.length == targetAmts.length, "invalid
lengths");
```

```
          requireValidCollArray(erc20s);

-         if (!reweightable && config.erc20s.length != 0) {
+         if ((!reweightable || (reweightable && !disableTargetAmountCheck)) &&
config.erc20s.length != 0) {
              // Require targets remain constant
              BasketLibP1.requireConstantConfigTargets(
                  assetRegistry,
@@ -238,20 +237,6 @@ contract BasketHandlerP1 is ComponentP1, IBasketHandler {
                  erc20s,
                  targetAmts
              );
-         } else if (normalize && config.erc20s.length != 0) {
-             // Confirm reference basket is SOUND
-             assetRegistry.refresh(); // will set lastStatus
-             require(lastStatus == CollateralStatus.SOUND, "unsound basket");
-
-             // Normalize targetAmts based on UoA value of reference basket, excl
issuance premium
-             (uint192 low, uint192 high) = price(false);
-             assert(low != 0 && high != FIX_MAX); // implied by SOUND status
-             targetAmts = BasketLibP1.normalizeByPrice(
-                 assetRegistry,
-                 erc20s,
-                 targetAmts,
-                 (low + high + 1) / 2
-             );
          }
```

Team response

Fixed by implementing the recommended approach. Documented that RTokens should use a spell for safety checks / normalization.

Mitigation review

Fixed as recommended.

Low severity findings

TRST-L-1 DAO fee in Distributor incurs large rounding losses for small percentages

- **Category:** Rounding issues
- **Source:** Distributor.sol
- **Status:** Acknowledged

Description

According to Reserve, the intended DAO fee percentages are 0% - 0.1% in the beginning. Later, they are supposed to increase to 5% - 10%.

For a fee percentage of 0.1%, it can be shown that the rounding error can be up to 10%. This means the DAO receives only 90% of the fees that it should receive.

The rounding loss can be explored in this graph.

Recommended mitigation

There is no easy solution for fixing the rounding error without a larger refactoring of *Distributor*. It is recommended to document the behavior. The percentage does not have to be exact.

Team response

Acknowledged. The behavior will be documented.

Mitigation review

The rounding loss has been documented correctly.

TRST-L-2 Changing distributions can temporarily affect the amount of funds sent to fee DAO

- **Category:** Logical issues
- **Source:** Distributor.sol
- **Status:** Acknowledged

Description

In the following scenario the effective fee rate can rise to 100% of the distributed funds for a single distribution cycle:

1. The RToken distributes a non-zero amount of RSR to its own revenue destination.
2. The RToken Governance removes its own RSR revenue destination such that only veRSR receives RSR.
3. Before the introduction of veRSR, any funds in the RSR *RevenueTrader* were sent back to *BackingManager* since **rsrTotal = 0**. But now, all the funds are distributed to veRSR.

Recommended mitigation

The issue cannot be addressed without a larger refactoring. It can be acknowledged, and Governance must be mindful of it when updating distributions.

Team response

Acknowledged. The incorrect distribution only occurs for a single distribution cycle. Distribution cycles are short, and therefore deal with limited amounts of funds.

Mitigation review

The finding has been acknowledged.

TRST-L-3 Division in GnosisTrade can cause precision loss and disable violation reporting

- **Category:** Rounding issues
- **Source:** GnosisTrade.sol
- **Status:** Fixed

Description

In *GnosisTrade*, the calculation of **worstCasePrice** and **clearingPrice** has been refactored.

Before the change:

```
uint192 clearingPrice = shiftl_toFix(adjustedBuyAmt, -int8(buy.decimals())).div(
        shiftl_toFix(adjustedSoldAmt, -int8(sell.decimals()))
);
```

After the change:

```
uint192 clearingPrice = divuu(adjustedBuyAmt, adjustedSoldAmt).shiftl(
    int8(sell.decimals()) - int8(buy.decimals()),
    FLOOR
);
```

The new formula is correct, but it causes issues when rounding is considered. For example, the token in the numerator can have 6 or 8 decimals and the token in the denominator up to 21 decimals.

divuu(adjustedBuyAmt, adjustedSoltAmt) could then perform a calculation like **1e8 * 1e18 / (1.1e5 * 1e21) = 0**.

This is the case where one whole token with 8 decimals (**1e8**) is bought for **110,000** whole tokens with 21 decimals (**1e5 * 1e21**). This is an edge case but within the allowed boundaries.

The impact of the rounding error is that the violation check for the trade is not performed correctly. In fact, when rounding results in **worstCasePrice = 0**, no violation can be detected.

Recommended mitigation

In communication with the client, it has been determined that the best mitigation is to remove decimal shifting of the sell and buy amounts and to introduce **1e9** additional precision to the calculation:

```
uint192 clearingPrice = shiftl_toFix(adjustedBuyAmt, 9).divu(adjustedSoldAmt, FLOOR);
```

The above solution doesn't lose precision due to shifting **adjustedBuyAmt** or **adjustedSoldAmt** to their 18 decimals representation, and intermediate multiplication by **1e9** ensures that even in an adverse scenario, the resulting **clearingPrice** is not meaningfully truncated.

Additionally, shifting **adjustedBuyAmt** left by 27 decimals does not introduce overflow concerns. The result of the shift must fit into **uint192**, i.e. it must be less than **6.2e57**. Considering the left shift by 27 decimals, **adjustedBuyAmt** must be less than about **6.2e30**. Given the requirement of $1 per 21 decimal token, this would require a value of $6.2 billion being traded in a single trade to overflow. Maximum trade volumes are orders of magnitudes lower.

Team response

Fixed.

Mitigation review

The recommendation has been implemented.

Additional recommendations

TRST-R-1 Rename parameters in IDistributor.setDistributions()

```
-    function setDistributions(address[] calldata dest, RevenueShare[] calldata share)
external;
+    function setDistributions(address[] calldata dests, RevenueShare[] calldata
shares) external;
```

TRST-R-2 Missing zero address check for feeRecipient in DAOFeeRegistry constructor

```
     constructor(address owner_) Ownable() {
+        if (owner_ == address(0)) revert DAOFeeRegistry__InvalidFeeRecipient();
         _transferOwnership(owner_); // Ownership to DAO
         feeRecipient = owner_; // DAO as initial fee recipient
```

TRST-R-3 Wrong documentation for Distributor.setDistributions() function

```
   /// @custom:governance
     // checks: invariants hold in post-state
     // effects:
-    //   destinations' = dests
-    //   distribution' = shares
+    //   destinations' = destinations.add(dests)
+    //   distribution' = distribution.set(dests[i], shares[i]) for i < dests.length
     function setDistributions(address[] calldata dests, RevenueShare[] calldata
shares)
```

TRST-R-4 Simplify condition in Distributor.distribute()

Below, **tokensPerShare * (totalShares - paidOutShares) > 0** can be simplified because **tokensPerShare > 0** is already required earlier in the function.
The **isRSR** check is also redundant since for **rToken**, **paidOutShares** will be equal **totalShares**.

```
         DAOFeeRegistry daoFeeRegistry = main.daoFeeRegistry();
         if (address(daoFeeRegistry) != address(0)) {
-            // DAO Fee
-            if (isRSR) {
+            if (totalShares > paidOutShares) {
                 (address recipient, , ) =
main.daoFeeRegistry().getFeeDetails(address(rToken));

-                if (recipient != address(0) && tokensPerShare * (totalShares -
paidOutShares) > 0) {
+                if (recipient != address(0)) {
                     IERC20Upgradeable(address(erc20)).safeTransferFrom(
                         caller,
                         recipient,
```

TRST-R-5 Broker.priceNotDecayed() should be renamed

In a previous audit, it has been determined that *Broker.priceNotDecayed()* should be renamed. The function doesn't check whether the price has decayed. Instead, it checks that the price has been updated at the current timestamp. Therefore, a more descriptive name is *pricedAtTimestamp()*.

TRST-R-6 DAOFeeRegistry can use RoleRegistry for access controls

DAOFeeRegistry, which has been introduced during the audit, inherits from *Ownable*. However, it is a cleaner solution to use *RoleRegistry* for its access controls which is already how *AssetPluginRegistry* and *VersionRegistry* manage their access controls.

TRST-R-7 upgradeMainTo() should check version after upgrading

Main.upgradeMainTo() will upgrade itself to the new *Main* implementation, and then upgrade other components in *upgradeRTokenTo()*.

upgradeMainTo(versionHash) does not check that the version of *Main* is as intended after the upgrade, but *upgradeRTokenTo(versionHash)* requires that the version of *Main* corresponds to **versionHash**.

It is recommended to check that the version of *Main* corresponds to **versionHash** after upgrading *Main* in *upgradeMainTo()*. That is, add the following check at the end of *upgradeMainTo()*:

```
    function upgradeMainTo(bytes32 versionHash) external onlyRole(OWNER) {
        require(address(versionRegistry) != address(0), "no registry");
        require(!versionRegistry.isDeprecated(versionHash), "version deprecated");

        Implementations memory implementation =
versionRegistry.getImplementationForVersion(
            versionHash
        );

        _upgradeProxy(address(this), address(implementation.main));
+       require(keccak256(abi.encodePacked(this.version())) == versionHash, "...");
    }
```

TRST-R-8 The _ensureSufficientTotal() check in Distributor.init() is incorrect

totals() will add DAO fee to **rsrTotal** in addition to shares in **distribution**. Since *_ensureSufficientTotal()* in *Distributor.init()* only checks for shares in **distribution** and does not consider DAO fee, this makes the check incorrect. The correct check should be as follows.

```
    function init(IMain main_, RevenueShare calldata dist) external initializer {
        __Component_init(main_);
        cacheComponents();
```

```
-       _ensureSufficientTotal(dist.rTokenDist, dist.rsrDist);
        _setDistribution(FURNACE, RevenueShare(dist.rTokenDist, 0));
        _setDistribution(ST_RSR, RevenueShare(0, dist.rsrDist));
+       RevenueTotals memory revTotals = totals();
+       _ensureSufficientTotal(revTotals.rTokenTotal, revTotals.rsrTotal);
    }
```

TRST-R-9 req.sellAmount > 1 check no longer ensures that sellAmount in tok units is greater zero for tokens with > 18 decimals

In *RevenueTrader.manageTokens()*, it is checked that **req.sellAmount > 1**. For tokens with **<= 18 decimals**, this implies that **sellAmount** in **{tok}** units in *DutchTrade* and *GnosisTrade* is also greater **1**. This is no longer the case, since by right-shifting **req.sellAmount** by up to 3 decimals, **sellAmount** can be zero.

It wasn't possible to find an impact to this behavior, the worst being that *_bidAmount()* can return **0** for dust amounts.

In addition, the issue can be considered nullified by how **req.sellAmount** is calculated.

```
req.sellAmount = s.shiftl_toUint(int8(trade.sell.erc20Decimals()), FLOOR);
```

As a result, **req.sellAmount** can only be non-zero if **s** is non-zero, and thus **sellAmount** in *DutchTrade* and *GnosisTrade* is non-zero.

For a mitigation, it can be useful to add an explicit **sellAmount > 0** check in both *DutchTrade* and *GnosisTrade*.

TRST-R-10 BasketHandler._quantity() can round in wrong direction

BasketHandler.quote() takes a rounding argument but the downstream *_quantity()* function has a hardcoded **CEIL** rounding parameter.

When quoting a redemption, **FLOOR** rounding should be used. *_quantity()* can incorrectly round up **1 wei**. The rounding error of **1 wei** is then scaled by the amount of basket units that are redeemed. For example, if 1 million basket units are redeemed, the rounding error becomes 1 million wei. All token amounts are represented in 18 decimals, so a rounding error of 1 million wei is almost certainly dust.

To mitigate the finding, *_quantity()* should accept a rounding direction as a parameter and perform its calculations according to this parameter. Redemptions should use **FLOOR** rounding.

Note that the *_price()* function also uses *_quantity()* to calculate a lower and an upper price. For the lower price, rounding in *_quantity()* should be **FLOOR**, and for the higher price the rounding should be **CEIL**. Calculating *_quantity()* twice may not be worth the overhead since asset prices have an error, so the rounding is negligible. The same argument can be made in other places like *basketsHeldBy()*, where the rounding is also not consequential enough to require a change.

TRST-R-11 Document that registered versions in VersionRegistry should include veRSR

By upgrading to version **>= 4.0.0** and setting a **versionRegistry**, an RToken commits itself to veRSR. It should not be possible for the RToken governance to escape the restrictions of veRSR. Consequently, veRSR must not register a version that lacks the veRSR restrictions. Effectively, these are versions **< 4.0.0**. This is a non-obvious requirement and should be documented.

TRST-R-12 Add explicit rounding direction for all calls to shiftl_toFix()

It is inconsistent that for some calls to *shiftl_toFix()*, the rounding direction is explicitly specified, for others it is not.

In the following instances, rounding mode should be specified as **FLOOR**:

- GnosisTrade.sol#L94
- CurveStableMetapoolCollateral.sol#L111
- CurveStableMetapoolCollateral.sol#L173
- CurveStableMetapoolCollateral.sol#L181
- CurveStableMetapoolCollateral.sol#L188
- YearnV2CurveFiatCollateral.sol#L67

In the following instance, rounding mode should be **CEIL**:

- ReadFacet.sol#L72

TRST-R-13 Call to BackingManager.forwardRevenue() leaves small amount of funds unprocessed due to rounding error

In *BackingManager.forwardRevenue()*, since **rTokenTotal + rsrTotal >= 1e4** is required, **>= 1e4** wei of tokens can be left undistributed when calculating **tokensPerShare**. For WBTC, which is a token with 8 decimals, this is about 6 USD. For the same reason, tokens will be left undistributed in **Distributor.distribute()**.

By multiplying the maximum number of revenue destinations by the maximum number of shares per revenue destination, it can be calculated that the maximum rounding loss per distribution is **10,000 * 100 * 2 = 2e6** wei. For WBTC, this would be about 1200 USD.

The undistributed tokens are then distributed according to the shares at the next time when the function is called, which is subject to the same rounding error again.

It is possible to calculate **tokensPerShare** with an increased precision to get rid of the rounding error. Practically, the rounding error can be considered negligible since no funds are lost and the revenue destination shares are most likely never set to the highest value, so implementing a higher precision is optional.

TRST-R-14 Consider forwarding revenue before setting new distributions

Since updating distributions with *Distributor.setDistribution()* or *Distributor.setDistributions()* changes the share of funds that are sent to the different revenue destinations, it can cause unexpected distribution of revenue if the revenue hasn't been processed before the update.

Consider calling *BackingManager.forwardRevenue()* before updating distributions to ensure existing revenue is distributed according to the old distribution table. If this is not feasible, additional documentation for the behavior can be added.

TRST-R-15 Implementation of AssetRegistry._reserveGas() can be more verbose

The current implementation of *AssetRegistry._reserveGas()* does not accurately reflect the check that it performs.

```
    function _reserveGas() private view returns (uint256) {
        uint256 gas = gasleft();
        require(
            gas > GAS_FOR_DISABLE_BASKET + GAS_FOR_BH_QTY,
            "not enough gas to unregister safely"
        );
        return gas - GAS_FOR_DISABLE_BASKET;
    }
```

What matters for security is that *basketHandler.quantity()* is called with at least **100k** gas, i.e., that *_reserveGas()* returns a value **>= 100k** and that this amount is actually passed on to *basketHandler.quantity()*.

This is necessary to avoid a situation where an attacker can provide a gas amount such that *basketHandler .quantity()* fails but the basket can still be disabled with the amount of gas that is available in the caller context due to the 63/64 rule.

The current code works correctly since **gas > GAS_FOR_DISABLE_BASKET + GAS_FOR_BH_QTY = 900k + 100k** is checked and the amount of gas with which *basketHandler.quantity()* is called is at least **1000k – 900k = 100k**. The 63/64 rule does not restrict the gas amount.

However, it is recommended to implement the *_reserveGas()* function in the following way:

```
    function _reserveGas() private view returns (uint256) {
        uint256 gas = gasleft();
        // Call to quantity() restricts gas that is passed along to 63 / 64 of
gasleft().
        // Therefore gasleft() must be greater than 64 * GAS_FOR_BH_QTY / 63
        // GAS_FOR_DISABLE_BASKET is a buffer which can be considerably lower without
        // security implications.
        require(
            gas > (64 * GAS_FOR_BH_QTY) / 63 + GAS_FOR_DISABLE_BASKET,
            "not enough gas to unregister safely"
        );
        return GAS_FOR_BH_QTY;
```

```
    }
```

This calls *basketHandler.quantity()* with an exact amount of gas, and makes the *require()* more expressive. **GAS_FOR_DISABLE_BASKET** is added for historic reasons and to be on the safe side. It's possible to make it considerably lower with sufficient testing. All that's needed is a small buffer in between the gas check in *_reserveGas()* and the restriction imposed by the 63/64 rule.

Centralization risks

TRST-CR-1 veRSR must be trusted

The 4.0.0 release allows the RToken Governance to set Version Registry, Asset Plugin Registry and DAO Fee Registry in *Main*.

RToken Governance is not required to set these addresses and can keep centralization risks and trust assumptions as prior to 4.0.0.

If the RToken Governance decides to assign contracts managed by veRSR to these registries, which is the intended use, and which cannot be undone, veRSR is granted the following permissions:

- veRSR can set the fee that it receives as a percentage of funds distributed in *Distributor* (maximum is 15%)
- veRSR must consent to any implementation upgrades by setting the implementations for versions that RToken Governance wants to upgrade to. Once implementations are set in the Version Registry, they cannot be unset. This makes it impossible for veRSR to maliciously front-run the upgrade and change the implementation. An RToken that has initiated an upgrade with *Main.upgradeMainTo()* can always finish the upgrade with *Main.upgradeRTokenTo()*. Versions can be deprecated either by veRSR or an emergency council that is set by veRSR.
- veRSR must consent to any asset plugins that RToken Governance wants to register. Assets can be registered and unregistered by veRSR and deprecated by the emergency council.

Note that different registry implementations can have additional consequences and the above privileges are assessed based on the audited registry implementations.

Overall, if an RToken Governance decides to set the registries to contracts that are controlled by veRSR, it introduces trust assumptions in both directions.

If veRSR acts maliciously, the RToken instance does not suffer an immediate loss of funds but is unable to replace collateral and unable to upgrade to new implementations. Both scenarios can cause a loss of funds as a secondary effect.

Systemic risks

TRST-SR-1 Issuance of RTokens is incentivized when collateral trades below peg

As described in TRST-M-2, issuance of RTokens is incentivized when collateral trades below its peg price. The attack surface was reduced significantly. Still, oracle errors can make it possible that RTokens are issued without providing the corresponding target amounts in market value, leading to an increased risk for RSR stakers.

And the issuance premium is optional, meaning there is no additional protection for RSR stakers when the issuance premium is disabled.

RSR stakers must assess the risk of each RToken that they stake in individually, based on the factors outlined in the description of the finding.