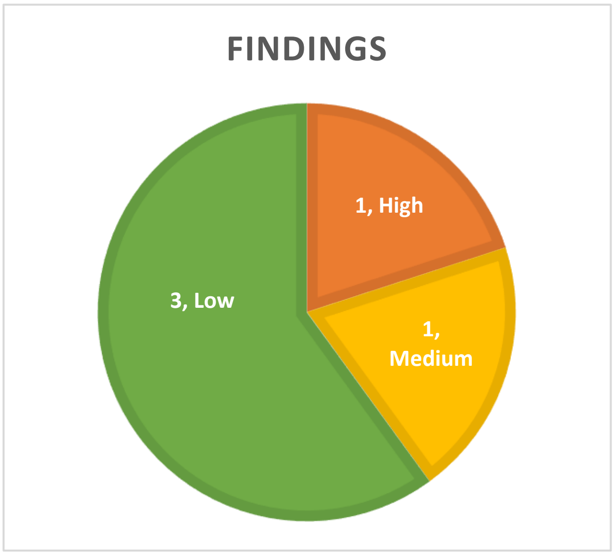
Trust Security

Smart Contract Audit

Reserve Protocol

23/02/24

Executive summary

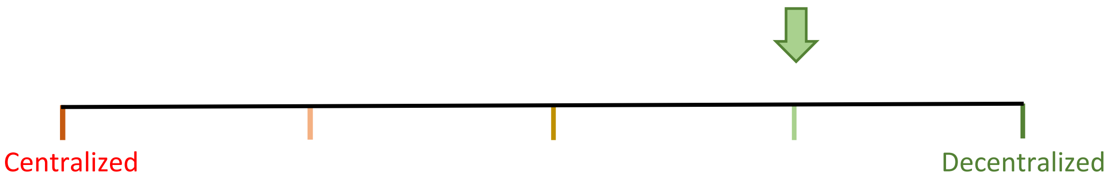


Category	Stablecoin
Audited file count	12
Lines of Code	1671
Auditor	cccz HollaDieWaldfee gjaldon
Time period	22/01/2024- 27/01/2024

Findings

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

Centralization score



Signature

Digitally signed by Trust
Date: 2024.02.23 12:08:08 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	27/01/2024	Client report
0.2	23/02/2024	Mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

The scope of the audit consists of changes to existing files since the previous Trust Security audit as well as new files.

- /contracts/plugins/trading/DutchTrade.sol
- /contracts/plugins/trading/DutchTradeRouter.sol
- /contracts/plugins/assets/frax/SFraxCollateral.sol
- /contracts/plugins/assets/frax-eth/vendor/CurvePoolEmaPriceOracleWithMinMax.sol
- /contracts/plugins/assets/frax-eth/SFraxEthCollateral.sol
- /contracts/plugins/assets/FraxOracleLib.sol
- /contracts/plugins/assets/stargate/StargateRewardableWrapper.sol
- /contracts/plugins/assets/yearnv2/YearnV2CurveFiatCollateral.sol
- /contracts/plugins/assets/morpho-aave/MorphoTokenisedDeposit.sol
- /contracts/p1/StRSR.sol
- /contracts/p1/BasketHandler.sol
- /contracts/p1/Deployer.sol

In addition, the integration of these changes with the codebase at large has been assessed as well.

Repository details

- **Repository URL:** https://github.com/reserve-protocol/protocol
- **Commit hash:** c6bbf00ae374a2fd1c7a970b91f4cbf514ea287c
- **Mitigation review hash:** b03cab62bde246d440a8c3c601567f341c559f07

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

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.

HollaDieWaldfee learnt his security skills in Web2 bug bounties which enabled him to quickly rise to the top of competitive audits, winning multiple contests. He is also a Senior Watson at Sherlock under his alias “roguereddwarf”.

Gjaldon is a DeFi specialist who enjoys numerical and economic incentives analysis. He transitioned to Web3 after 10+ years working as a Web2 engineer. His first foray into Web3 was achieving first place in a smart contracts hackathon and then later securing a project grant to write a contract for Compound III. He shifted to Web3 security and in 3 months achieved top 2-5 in two contests with unique High and Medium findings and joined exclusive top-tier auditing firms.

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.

Qualitative analysis

Metric	**Rating**	**Comments**
Code complexity	**Excellent**	Project kept code as simple as possible, reducing attack risks
Documentation	**Good**	Project is mostly very well documented.
Best practices	**Good**	Project generally follows best practices.
Centralization risks	**Good**	If system parameters are chosen safely, centralized is mostly minimized.

Findings

High severity findings

TRST-H-1 Incorrect calculation in YearnV2CurveFiatCollateral._pricePerShare()

- **Category:** Logical flaws
- **Source:** YearnV2CurveFiatCollateral.sol
- **Status:** Fixed

Description

Using **yvCurveUSDCcrvUSD** as an example, *YearnV2CurveFiatCollateral._pricePerShare()* is used to calculate how much **crvUSDUSDC-f** each **yvCurveUSDCcrvUSD** is worth.

_pricePerShare() calls *pricePerShareHelper.amountToShares(),* when calling *pricePerShareHelper.amountToShare()*, the parameter **amount** passed into *amountToShares()* is *vault.totalSupply().*

```
    function _pricePerShare() internal view returns (uint192) {
        uint256 supply = erc20.totalSupply(); // {qTok}
        uint256 shares = pricePerShareHelper.amountToShares(address(erc20), supply);
// {qLP Token}

        // yvCurve tokens always have the same number of decimals as the underlying
curve LP token,
        // so we can divide the quanta units without converting to whole units

        // {LP token/tok} = {LP token} / {tok}
        return divuu(shares, supply);
    }
```

Therefore, the calculation in *_pricePerShare()* is **supply * supply / totalAssets / supply = supply / totalAssets,** which is incorrect. The *_pricePerShare()* calculation should be **totalAssets / supply** (this is actually the reciprocal of *_pricePerShare()*).

```
    function amountToShares(address vault, uint amount) external view returns (uint) {
        uint totalSupply = IVault(vault).totalSupply();
        if (totalSupply > 0) {
            return amount * totalSupply / calculateFreeFunds(vault);
        }
        return amount;
    }
```

This issue causes *_pricePerShare()* to be smaller than its actual value, since **totalAssets / supply** is greater than one. The protocol's valuation of the collateral is thus smaller. Taking **crvUSDUSDC-f** as an example, 0.96 will be used instead of 1.04. So the price will be estimated to be only 92.3% of its true value, and this percentage drops over time.

Recommended mitigation

It is recommended to call *pricePerShareHelper.sharesToAmount()* instead of *pricePerShareHelper.amountToShares()* in *_pricePerShare().*

```
diff --git a/contracts/plugins/assets/yearnv2/YearnV2CurveFiatCollateral.sol
b/contracts/plugins/assets/yearnv2/YearnV2CurveFiatCollateral.sol
index 322eca9..a2012f3 100644
--- a/contracts/plugins/assets/yearnv2/YearnV2CurveFiatCollateral.sol
+++ b/contracts/plugins/assets/yearnv2/YearnV2CurveFiatCollateral.sol
@@ -98,12 +98,12 @@ contract YearnV2CurveFiatCollateral is CurveStableCollateral {
     /// @return {LP token/tok}
     function _pricePerShare() internal view returns (uint192) {
         uint256 supply = erc20.totalSupply(); // {qTok}
-        uint256 shares = pricePerShareHelper.amountToShares(address(erc20), supply);
// {qLP Token}
+        uint256 amounts = pricePerShareHelper.sharesToAmount(address(erc20), supply);
// {qLP Token}

         // yvCurve tokens always have the same number of decimals as the underlying
curve LP token,
         // so we can divide the quanta units without converting to whole units

         // {LP token/tok} = {LP token} / {tok}
-        return divuu(shares, supply);
+        return divuu(amounts, supply);
     }
 }
```

It is also possible to call *erc20.pricePerShare()* and handle the decimals.

Team response

Fixed.

Mitigation Review

This issue is fixed as recommended. Note that *_pricePerShare()* can be optimized based on the following observation.

```
The calculation is currently:

totalAssets = sharesToAmount(totalSupply)
return 1e18 * totalAssets / totalSupply
<->
totalAssets = totalSupply * freeFunds / totalSupply
return 1e18 * totalAssets / totalSupply
<->
return 1e18 * totalSupply * freeFunds / totalSupply / totalSupply
<->
return 1e18 * freeFunds / totalSupply
```

This is equivalent to calling *sharesToAmount(1e18)*.

```
diff --git a/contracts/plugins/assets/yearnv2/YearnV2CurveFiatCollateral.sol
b/contracts/plugins/assets/yearnv2/YearnV2CurveFiatCollateral.sol
index d3b0d1b8..ff85152e 100644
--- a/contracts/plugins/assets/yearnv2/YearnV2CurveFiatCollateral.sol
+++ b/contracts/plugins/assets/yearnv2/YearnV2CurveFiatCollateral.sol
@@ -100,13 +100,6 @@ contract YearnV2CurveFiatCollateral is CurveStableCollateral {

     /// @return {LP token/tok}
     function _pricePerShare() internal view returns (uint192) {
-        uint256 supply = erc20.totalSupply(); // {qTok}
-        uint256 amount = pricePerShareHelper.sharesToAmount(address(erc20), supply);
// {qLP Token}
-
-        // yvCurve tokens always have the same number of decimals as the underlying
curve LP token,
-        // so we can divide the quanta units without converting to whole units
-
-        // {LP token/tok} = {LP token} / {tok}
-        return divuu(amount, supply);
+        return _safeWrap(pricePerShareHelper.sharesToAmount(address(erc20),
FIX_ONE));
     }
 }
```

Medium severity findings

TRST-M-1 Revenue and backing buffer can be lost when changing target amounts

- **Category:** Logical flaws
- **Source:** BasketHandler.sol, RToken.sol
- **Status:** Open

Description

The **reweightable** flag has been added which, when enabled, allows changing the target amounts in the prime basket configuration.

As a side effect, this can allow RToken holders to redeem an old basket via *RToken.redeemCustom()* and steal part of the revenue and backing buffer.

Let's consider the case where the prime basket contains two target amounts.

```
Assume we start with a prime basket with:
targetAmount[BTC] = b1 = 1
targetAmount[ETH] = e1 = 0

Now we reconfigure (rebalance) the prime basket and have:
targetAmount[BTC] = b2 = 0.9
targetAmount[ETH] = e2 = 1.8 (since 0.1 BTC = 1.8 ETH)

Assume the RToken is fully collateralized in the new configuration and we have a
overcollateralization (backing buffer + revenue) of x (e.g., 0.05 = 5%).

If 1 basket is redeemed with the new configuration the result is:
    -   0.9 BTC
    -   1.8 ETH

If 1 basket is redeemed with the old configuration the result is:
    -   y BTC with y = max(b1,(b2 + x)) BTC (capped at b1 which is the old basket
        configuration)
    -   e1 ETH

The question is what is the price p (ETH / BTC) when it becomes more profitable to
redeem for the old basket (i.e., to eat up backing buffer / revenue) given some x?

It is when
((y - b2) BTC * p > (e2 - e1) ETH)
<-> (p > (e2 - e1) ETH / (y - b2) BTC)

Let's say backing buffer is 5%, i.e., x = 0.05.
```

```
Then: p > 1.8 ETH / 0.05 BTC = 36 ETH / BTC.

It’s noticeable that the smaller the rebalancing, the less p needs to change (worst case
is the immediate loss as soon as the price moves when the backing buffer allows to
redeem the full old basket):

Let's change the basket to b2 = 0.99 BTC and e2 = 0.18 ETH instead and x = 0.01.

Then we get p > 0.18 ETH / (1 BTC - 0.99 BTC) = 18 ETH / BTC.

This means that immediately when the ETH / BTC price starts moving up, it's more
profitable to eat up backing buffer / revenue rather than to redeem the new basket.
For example, when price is 19 ETH / BTC then by redeeming for the old basket we get 0.01
BTC * 19 ETH / BTC = 0.19 ETH vs. 0.18 ETH if we redeem for the new basket.
```

Based on these calculations, if we have an "index" RToken that performs small rebalancing steps we get a rainout of the backing buffer / revenue.

The difference between 18 ETH / BTC and 19 ETH /BTC can even be a 1-day price movement.

This demonstrates that limiting redemptions for an old basket based on a time threshold like 1 week is an insufficient fix. Redemptions for the old basket should be stopped as soon as possible and given that the purpose of custom redemptions is just to smoothen the recollateralization process, redemptions for the old basket should be stopped once recollateralization is over.

During recollateralization, the effect of such an adverse price movement is offset partially or fully by the fact that some or all of the "to-be-traded assets" are still in the old target and appreciate if the old target appreciates.

Also note that the conclusions generalize to more complicated baskets that have more than two target amounts. In fact, the more general case tends to be less severe since a change in the relative price for one target can be offset by the change in another target (and the gain from a favorable move is capped by the backing buffer + revenue while the loss from an adverse move can generally be bigger).

```
Example:
old basket
targetAmount[BTC] = 1
targetAmount[ETH] = 0
targetAmount[USD] = 0

->
new basket
targetAmount[BTC] = 0.8
targetAmount[ETH] = 1.8
targetAmount[USD] = 4000
```

If an attacker wants to maximize USD and BTC appreciates against USD, then this can be offset by ETH appreciating against BTC. In other words, the general case can be at most as profitable for the attacker as the case with two target amounts.

Another area of concern with changing the target amounts is the delay between proposing a basket change and executing the proposal.

When the proposal is executed, the RToken can immediately be undercollateralized (new target has appreciated against the old target) and **RSR** stakers get slashed. Or the RToken can end up overcollateralized and RToken holders lose part of their funds (new target has depreciated against the old target).

Recommended mitigation

For the issue with diverging target values after the proposal is executed, we recommend disabling redemptions for the old basket when the RToken has reached full collateralization after a basket change (optionally this may depend on **reweightable = true**). Thereby custom redemptions are possible for at most one historical basket at a time.

For the changing target values during the proposal stage, a minimal solution is to rely on the **guardian** to cancel proposals when the target values have diverged too much.

A better solution is to perform this check within the smart contracts and reject any prime basket change for which the value of the new prime basket has diverged too much from the value of the current basket. The threshold for this check could be a mutable governance parameter.

Note that the protocol must be able to handle cases where the current basket pricing is corrupted so the *BasketHandler.setPrimeBasket()* function may include an additional parameter that can enable or disable the threshold check such as not to introduce new dependencies to *setPrimeBasket()*.

As discussed with the client, comparing the basket values can also decrease the attack surface where an RToken has sufficient buying power to move collateral prices in anticipation of the governance proposal executing.

Team response

Two-part fix consisting of a and b.

Mitigation Review

There are three issues that should be considered.

1. The *BasketLib.normalizeByPrice()* function doesn't check the collateral status (SOUND, IFFY, DISABLED). It only checks that the price is within **(0,FIX_MAX)**.

This is insufficient as the pricing of the old basket requires that all its collaterals are SOUND. So, the same requirement should be applied for the new collaterals as well.

```
+ require(coll.status() == CollateralStatus.SOUND);
```

2. *BasketHandler.trackCollateralization()* is only called from within *BackingManager.settleTrade()*. Invalidating old basket nonces in this way only works if the basket switch results in trades being performed and the basket becomes fully collateralized as part of settling the trade.
If the *BackingManager* just takes a haircut without performing a trade (or takes a haircut non-atomically with settling a trade as in *GnosisTrade* trades) no basket nonces are invalidated.
Similarly, if a basket switch doesn't result in a trade, no basket nonces are invalidated.
While old basket nonces can be invalidated by anyone calling the

BasketHandler.trackCollateralization() function directly, it would be better to make this part of the protocol logic and not to rely on external actors.
This is difficult since anyone can send tokens to the *BackingManager* at any time and shortcut the recollateralization process. It is not possible to capture all state transitions from fully collateralized to not fully collateralized and vice versa since no callback can be executed on these transitions. So, any solution by design is only an approximation.

The issue can be addressed by invalidating as many **nonces** as possible even if the current basket is not fully collateralized.

```
    function trackCollateralization() external {
+       assetRegistry.refresh();
        if (nonce > lastCollateralized && fullyCollateralized()) {
            emit LastCollateralizedChanged(lastCollateralized, nonce);
            lastCollateralized = nonce;
+       } else if (nonce > 1 && nonce - 1 > lastCollateralized) {
+           emit LastCollateralizedChanged(lastCollateralized, nonce - 1);
+           lastCollateralized = nonce - 1;
+       }
```

The BasketHandler.*trackCollateralization()* function should also be called inside *RToken.redeemCustom()*.

However, this proposed fix assumes that the basket at **nonces – 1** has been fully collateralized.

3. *AssetRegistry.refresh()* should be called to get the latest collateral information before *fullyCollateralized()* is called, otherwise the result of the downstream *basketsHeldBy()* function will be stale.

```
    function trackCollateralization() external {
+       assetRegistry.refresh();
        if (nonce > lastCollateralized && fullyCollateralized()) {
            emit LastCollateralizedChanged(lastCollateralized, nonce);
            lastCollateralized = nonce;
        }
    }
```

Similarly, *refresh()* is not called before the *status()* check in *_registerIgnoringCollisions()*.

```
diff --git a/contracts/p1/AssetRegistry.sol b/contracts/p1/AssetRegistry.sol
index 2d606999..9c0dcaf2 100644
--- a/contracts/p1/AssetRegistry.sol
+++ b/contracts/p1/AssetRegistry.sol
@@ -190,6 +190,8 @@ contract AssetRegistryP1 is ComponentP1, IAssetRegistry {
     // effects: assets' = assets.set(asset.erc20(), asset)
     // returns: assets[asset.erc20()] != asset
     function _registerIgnoringCollisions(IAsset asset) private returns (bool swapped)
{
+        // Refresh to ensure it does not revert, and to save a recent lastPrice
+        asset.refresh();
         if (asset.isCollateral()) {
             require(
                 ICollateral(address(asset)).status() == CollateralStatus.SOUND,
@@ -208,9 +210,6 @@ contract AssetRegistryP1 is ComponentP1, IAssetRegistry {
         assets[erc20] = asset;
         emit AssetRegistered(erc20, asset);
```

```
-        // Refresh to ensure it does not revert, and to save a recent lastPrice
-        asset.refresh();
-
         if (!main.frozen()) {
             backingManager.grantRTokenAllowance(erc20);
         }
```

Low severity findings

TRST-L-1 StargateRewardableWrapper can revert on withdrawals

- **Category:** Logical flaws
- **Source:** StargateRewardableWrapper.sol
- **Status:** Fixed

Description

The recommendations in the TRST-L-6 finding in the September 2023 audit were provided based on the knowledge that was available at the time.

Only the Mainnet deployment of Stargate LP Staking was referenced in the codebase and so the *StargateRewardableWrapper* was refactored to optimize it for the Mainnet deployment.

Meanwhile, the *StargateRewardableWrapper* has been changed to also work with the BASE deployment of Stargate LP Staking which, while generally implementing the same calculations, has some small differences:

1. Trying to claim rewards that exceed the contract's balance causes a revert (here, line 180)
2. The case when **totalAllocPoint == 0** does not cause a revert (here, line 148)

This finding aims to solve a "Low" severity issue in *StargateRewardableWrapper* that occurs when the *withdraw()* function is called on the BASE deployment and the reward balance is insufficient. The *StargateRewardableWrapper* would fail to withdraw the funds from the Stargate LP Staking contract, resulting in the inability for some users to withdraw from the wrapper. Being able to withdraw from the wrapper should have priority over not missing out on rewards. Note that the new behavior with the recommended changes applied could be used in a griefing attack by someone triggering a withdrawal with the purpose of giving up on rewards.

Recommended mitigation

To solve the issue, the error should be caught and *emergencyWithdraw()* should be called which bypasses the reward payout.

In addition, the optimizations for the Mainnet deployment should be removed and the *StargateRewardableWrapper* should implement the minimal code to be able to manage both deployments.

```
diff --git a/contracts/plugins/assets/stargate/StargateRewardableWrapper.sol
b/contracts/plugins/assets/stargate/StargateRewardableWrapper.sol
index 16136eff..84d57cc5 100644
--- a/contracts/plugins/assets/stargate/StargateRewardableWrapper.sol
```

```
+++ b/contracts/plugins/assets/stargate/StargateRewardableWrapper.sol
@@ -48,42 +48,30 @@ contract StargateRewardableWrapper is RewardableERC20Wrapper {
     }
 
     function _claimAssetRewards() internal override {
-        try stakingContract.totalAllocPoint() returns (uint256 totalAllocPoint) {
-            if (totalAllocPoint == 0) {
-                return;
-            }
-        } catch {
-            return;
-        }
-
         // `.deposit` call in a try/catch to prevent staking contract
         // this is because `_claimAssetRewards` is called on all movements
         // and we want to prevent external calls from bricking the contract
+        // the call can also fail for the Mainnet deployment when totalAllocPoint ==
0
         // solhint-disable-next-line no-empty-blocks
         try stakingContract.deposit(poolId, 0) {} catch {}
     }
 
     function _afterDeposit(uint256, address) internal override {
         uint256 underlyingBalance = underlying.balanceOf(address(this));
-        try stakingContract.poolInfo(poolId) returns (IStargateLPStaking.PoolInfo
memory poolInfo) {
-            if (poolInfo.allocPoint != 0 && underlyingBalance != 0) {
-                pool.approve(address(stakingContract), underlyingBalance);
-                try stakingContract.deposit(poolId, underlyingBalance) {} catch {}
-            }
-        } catch {}
+        IStargateLPStaking.PoolInfo memory poolInfo =
stakingContract.poolInfo(poolId);
+        if (poolInfo.allocPoint != 0 && underlyingBalance != 0) {
+            pool.approve(address(stakingContract), underlyingBalance);
+            try stakingContract.deposit(poolId, underlyingBalance) {} catch {}
+        }
     }
 
     function _beforeWithdraw(uint256 _amount, address) internal override {
-        try stakingContract.poolInfo(poolId) returns (IStargateLPStaking.PoolInfo
memory poolInfo) {
-            uint256 underlyingBalance = underlying.balanceOf(address(this));
+        uint256 underlyingBalance = underlying.balanceOf(address(this));
 
-            if (underlyingBalance < _amount) {
-                if (poolInfo.allocPoint != 0) {
-                    try stakingContract.withdraw(poolId, _amount - underlyingBalance)
{} catch {}
-                } else {
-                    try stakingContract.emergencyWithdraw(poolId) {} catch {}
-                }
+        if (underlyingBalance < _amount) {
+            try stakingContract.withdraw(poolId, _amount - underlyingBalance) {}
catch {
+                // emergencyWithdraw() is always successful for both Mainnet and BASE
deployments
+                stakingContract.emergencyWithdraw(poolId);
             }
-        } catch {}
-    }
+        }
 }
```

Team response

Fixed.

Mitigation Review

The recommendation has been implemented. In addition, it is recommended to revert in the **catch** blocks **if (errData.length == 0)** to follow the practice used throughout the codebase and prevent griefing attacks.

TRST-L-2 SFraxCollateral and SFraxEthCollateral should override refresh() and sync rewards

- **Category:** Logical flaws
- **Source:** SFraxCollateral.sol, SFraxEthCollateral.sol
- **Status:** Fixed

Description

Both underlying Vaults of *SFraxCollateral* (sFRAX) and *SFraxEthCollateral* (sfrxETH) pay out rewards over multiple cycles and it is correct that rewards within the current cycle are automatically accounted for as part of the Vaults' *totalAssets()* function.

However, new cycles are only entered and accounted for by calling a non-view function.

Both Vaults are currently in active use and daily updates are to be expected. Still, it is beneficial to perform the update as part of *refresh()* in order to not rely on other users moving the Vaults into new reward cycles.

Recommended mitigation

Call *IsfrxEth.syncRewards()* to sync rewards for *SFraxEthCollateral*:

```
diff --git a/contracts/plugins/assets/frax-eth/SFraxEthCollateral.sol
b/contracts/plugins/assets/frax-eth/SFraxEthCollateral.sol
index ccafd116..b187feb3 100644
--- a/contracts/plugins/assets/frax-eth/SFraxEthCollateral.sol
+++ b/contracts/plugins/assets/frax-eth/SFraxEthCollateral.sol
@@ -41,6 +41,11 @@ contract SFraxEthCollateral is AppreciatingFiatCollateral,
CurvePoolEmaPriceOrac
         require(config.defaultThreshold > 0, "defaultThreshold zero");
     }

+    function refresh() public virtual override {
+        IsfrxEth(address(erc20)).syncRewards();
+        super.refresh();
+    }
+
     /// Can revert, used by other contract functions in order to catch errors
     /// @return low {UoA/tok} The low price estimate
     /// @return high {UoA/tok} The high price estimate
```

Call *IStakedFrax.syncRewardsAndDistribution()* to sync rewards for *SFraxCollateral*:

```
diff --git a/contracts/plugins/assets/frax/SFraxCollateral.sol
b/contracts/plugins/assets/frax/SFraxCollateral.sol
index 90b73318..a441fd50 100644
--- a/contracts/plugins/assets/frax/SFraxCollateral.sol
+++ b/contracts/plugins/assets/frax/SFraxCollateral.sol
```

```
@@ -25,6 +25,11 @@ contract SFraxCollateral is AppreciatingFiatCollateral {
         AppreciatingFiatCollateral(config, revenueHiding)
     {}
 
+    function refresh() public virtual override {
+        IStakedFrax(address(erc20)).syncRewardsAndDistribution();
+        super.refresh();
+    }
+
     // solhint-enable no-empty-blocks
 
     /// @return {ref/tok} Actual quantity of whole reference units per whole
collateral tokens
```

Team response

Fixed.

Mitigation Review

The recommendation has been implemented, and the external calls have been wrapped in **try-catch** blocks. In addition, it is recommended to revert in the **catch** block to follow best practice and prevent griefing attacks as in TRST-L-1.

TRST-L-3 Exchange rate calculations reverting can brick the system

- **Category:** Logical flaws
- **Source:** SFraxCollateral.sol, YearnV2CurveFiatCollateral.sol, AppreciatingFiatCollateral.sol
- **Status:** Fixed

Description

The calculation for *SFraxCollateral*'s exchange rate is implemented as shown below:

```
    function _underlyingRefPerTok() internal view override returns (uint192) {
        return
            divuu(
                IStakedFrax(address(erc20)).totalAssets(),
                IStakedFrax(address(erc20)).totalSupply()
            );
    }
```

When the *SFrax* external contract has a total supply of 0, the calculation will revert with a division-by-zero error. Given that *SFraxCollateral* inherits from *AppreciatingFiatCollateral*, its *refresh()* function would revert since it calls *_underlyingRefPerTok()*.

Every time a registered asset's *refresh()* reverts, core functionalities of the protocol are frozen since they refresh all assets registered in the system. Below is a list of some of these critical functions:

1. Redemptions of RTokens
2. Issuance of RTokens
3. Forwarding of Revenue

4. Rebalancing

In effect, the system is unavailable until the faulty collateral is unregistered. Note that all collaterals that inherit from *AppreciatingFiatCollateral* can have this effect.

Recommended mitigation

Modify *refresh()* in the *AppreciatingFiatCollateral* so that the call to *_underlyingRefPerTok()* is wrapped in a try-catch clause.

```
diff --git a/contracts/plugins/assets/AppreciatingFiatCollateral.sol
b/contracts/plugins/assets/AppreciatingFiatCollateral.sol
index 4e4c6a75..d0f93063 100644
--- a/contracts/plugins/assets/AppreciatingFiatCollateral.sol
+++ b/contracts/plugins/assets/AppreciatingFiatCollateral.sol
@@ -83,7 +83,16 @@ abstract contract AppreciatingFiatCollateral is FiatCollateral {
         // must happen before tryPrice() call since `refPerTok()` returns a stored
value

         // revenue hiding: do not DISABLE if drawdown is small
-        uint192 underlyingRefPerTok = _underlyingRefPerTok();
+        uint192 underlyingRefPerTok;
+
+        try this.getUnderlyingRefPerTok() returns (uint192 _underlyingRefPerTok) {
+            underlyingRefPerTok = _underlyingRefPerTok;
+        } catch (bytes memory errData) {
+            if (errData.length == 0) revert(); // solhint-disable-line reason-string
+            markStatus(CollateralStatus.IFFY);
+            emitStatusChange(oldStatus);
+            return;
+        }

         // {ref/tok} = {ref/tok} * {1}
         uint192 hiddenReferencePrice = underlyingRefPerTok.mul(revenueShowing); //
95% to protect against exchange rates going down by 5% since some assets have special
cases when their exchange rates drop
@@ -125,6 +134,10 @@ abstract contract AppreciatingFiatCollateral is FiatCollateral {
             markStatus(CollateralStatus.IFFY);
         }

+        emitStatusChange(oldStatus);
+    }
+
+    function getUnderlyingRefPerTok() public view returns (uint192) {
+        return _underlyingRefPerTok();
+    }
+
+    function emitStatusChange(CollateralStatus oldStatus) internal {
         CollateralStatus newStatus = status();
         if (oldStatus != newStatus) {
             emit CollateralStatusChanged(oldStatus, newStatus);
```

SFraxCollateral._underlyingRefPerTok() can also be modified to address the possible division-by-zero error.

```
diff --git a/contracts/plugins/assets/frax/SFraxCollateral.sol
b/contracts/plugins/assets/frax/SFraxCollateral.sol
index 9aea696e..9e3867fd 100644
--- a/contracts/plugins/assets/frax/SFraxCollateral.sol
+++ b/contracts/plugins/assets/frax/SFraxCollateral.sol
@@ -28,15 +28,8 @@ contract SFraxCollateral is AppreciatingFiatCollateral {
     // solhint-enable no-empty-blocks
     /// @return {ref/tok} Actual quantity of whole reference units per whole
collateral tokens
-    function _underlyingRefPerTok() internal view override returns (uint192) {
-        return
-            divuu(
-                IStakedFrax(address(erc20)).totalAssets(),
-                IStakedFrax(address(erc20)).totalSupply()
-            );
+    /// @return _rate {ref/tok} Quantity of whole reference units per whole
collateral tokens
+    function _underlyingRefPerTok() internal view virtual override returns (uint192)
{
+        return address(erc20).convertToAssets(1e18);
     }
 }
```

Depending on the fix for TRST-H-1, a division-by-zero check should also be applied in *YearnV2CurveFiatCollateral*.

Team response

Fixed.

Mitigation Review

The recommended division-by-zero check for *SFraxCollateral* and *YearnV2CurveFiatCollateral* hasn't been implemented. It is an extreme edge case for either of the total supplies to be zero. And instead of reverting and blocking the protocol completely, the collateral will now be disabled. No further change is needed.

With regards to wrapping the call to *_underlyingRefPerTok()* in a **try-catch** block, the recommendation has been implemented with the change that the status is set to **DISABLED** instead of **IFFY** if *underylingRefPerTok()* reverts. This is the better choice since a revert in *underylingRefPerTok()* by definition constitutes a complete failure of the collateral plugin and so the plugin doesn't need to be given the chance to recover.

Additional recommendations

MorphoTokenisedDeposit reward period can be griefed

In response to the previous mitigation review for the TRST-H-2 finding, the reward accounting in *MorphoTokenisedDeposit* has been changed such that now the **remainingPeriod** is only reset to **PAYOUT_PERIOD** if the contract detects new rewards.

This introduces a griefing concern whereby a user could deliberately claim rewards (or send MORPHO once the token becomes transferable) to lengthen the effective reward period as described in the previous mitigation review.

The MORPHO reward schedule is not well documented and so it may be possible that rewards cannot be claimed twice within the 7-day **PAYOUT_PERIOD** which would as mentioned only make this finding a concern once MORPHO becomes transferrable.

A possible solution is to only reset **remainingPeriod** once it has reached zero. This would allow for a more deterministic payout schedule while the downside is that rewards that become available while **remainingPeriod > 0** are delayed.

```
diff --git a/contracts/plugins/assets/morpho-aave/MorphoTokenisedDeposit.sol
b/contracts/plugins/assets/morpho-aave/MorphoTokenisedDeposit.sol
index e2bf558f..243d88ab 100644
--- a/contracts/plugins/assets/morpho-aave/MorphoTokenisedDeposit.sol
+++ b/contracts/plugins/assets/morpho-aave/MorphoTokenisedDeposit.sol
@@ -65,7 +65,7 @@ abstract contract MorphoTokenisedDeposit is RewardableERC4626Vault {
             state.availableBalance += amtToPayOut;
         }

-        if (newlyAccumulated != 0) {
+        if (newlyAccumulated != 0 && state.remainingPeriod <= timeDelta) {
             state.totalAccumulatedBalance = totalAccumulated;
             state.pendingBalance += newlyAccumulated;
```

Employ preventive measure against reentrancy in DutchTrade.settle()

The *bidWithCallback()* function has been added to *DutchTrade*. And while there is currently no reentrancy risk, the contract is prone to becoming vulnerable in the future as a result of changing existing contracts or adding new trade contracts.

The following sequence of calls can potentially cause accounting errors during recollateralization:

1. *DutchTrade.bidWithCallback()* is called and the attacker gets a callback. Note that at this point the **amountIn** has not yet been transferred to the *DutchTrade* contract.
2. The attacker calls *BackingManager.settleTrade()* to settle the *DutchTrade.*
3. The attacker opens a new trade (of a different kind) in the *BackingManager*. Note that a wrong or inefficient trade may be opened since **amountIn** is still missing.
4. If it was possible to get the new trade into a state where it can be settled within the same block, the attack can succeed and once execution resumes in

DutchTrade.bidWithCallback(), the new trade can be settled in the call to *origin.settleTrade().*

It is recommended to check that the *DutchTrade* is not settled from within the callback:

```
diff --git a/contracts/plugins/trading/DutchTrade.sol
b/contracts/plugins/trading/DutchTrade.sol
index 972aeecf..dca73917 100644
--- a/contracts/plugins/trading/DutchTrade.sol
+++ b/contracts/plugins/trading/DutchTrade.sol
@@ -116,6 +116,8 @@ contract DutchTrade is ITrade {
     address public bidder;
     // the bid amount is just whatever token balance is in the contract at settlement
time

+    bool public inCallback;
+
     // This modifier both enforces the state-machine pattern and guards against
reentrancy.
     modifier stateTransition(TradeStatus begin, TradeStatus end) {
         require(status == begin, "Invalid trade state");
@@ -266,7 +268,9 @@ contract DutchTrade is ITrade {
         sell.safeTransfer(bidder, lot()); // {qSellTok}

         uint256 balanceBefore = buy.balanceOf(address(this)); // {qBuyTok}
+        inCallback = true;
         IDutchTradeCallee(bidder).dutchTradeCallback(address(buy), amountIn, data);
+        inCallback = false;
         require(
             amountIn <= buy.balanceOf(address(this)) - balanceBefore,
             "insufficient buy tokens"
@@ -289,6 +293,7 @@ contract DutchTrade is ITrade {
     {
         require(msg.sender == address(origin), "only origin can settle");
         require(bidder != address(0) || block.number > endBlock, "auction not over");
+        require(!inCallback, "auction in callback");

         if (bidType == BidType.CALLBACK) {
             soldAmt = lot(); // {qSellTok}
```

Improve documentation of reweightable variable

The documentation of the **reweightable** variable in the **DeploymentParams** struct is unclear. Consider improving it, for example:

```
diff --git a/contracts/interfaces/IDeployer.sol b/contracts/interfaces/IDeployer.sol
index d811aef2..535a5d4f 100644
--- a/contracts/interfaces/IDeployer.sol
+++ b/contracts/interfaces/IDeployer.sol
@@ -38,7 +38,7 @@ struct DeploymentParams {
     //
     // === BasketHandler ===
     uint48 warmupPeriod; // {s} how long to wait until issuance/trading after
regaining SOUND
-    bool reweightable; // whether the basket can change in value
+    bool reweightable; // whether the target amounts in the prime basket can change
     //
     // === BackingManager ===
     uint48 tradingDelay; // {s} how long to wait until starting auctions after
switching basket
```

Document the expected erc20 for *YearnV2CurveFiatCollateral*

YearnV2CurveFiatCollateral inherits from *CurveStableCollateral* which expects its **erc20** to be a *RewardableERC20*.

```
    /// @dev config.erc20 should be a RewardableERC20
    constructor(
        CollateralConfig memory config,
        uint192 revenueHiding,
        PTConfiguration memory ptConfig
    ) AppreciatingFiatCollateral(config, revenueHiding) PoolTokens(ptConfig) {
        require(config.defaultThreshold > 0, "defaultThreshold zero");
    }
```

Even though *YearnV2CurveFiatCollateral* inherits from *CurveStableCollateral*, it expects its **erc20** to be a Yearn Vault. It is recommended to document this expectation to minimize confusion.

Mitigation Review

The following changes to the comments need to be applied.

```
diff --git a/contracts/plugins/assets/yearnv2/YearnV2CurveFiatCollateral.sol
b/contracts/plugins/assets/yearnv2/YearnV2CurveFiatCollateral.sol
index d3b0d1b8..b1b4ae9d 100644
--- a/contracts/plugins/assets/yearnv2/YearnV2CurveFiatCollateral.sol
+++ b/contracts/plugins/assets/yearnv2/YearnV2CurveFiatCollateral.sol
@@ -33,7 +33,7 @@ contract YearnV2CurveFiatCollateral is CurveStableCollateral {
     IPricePerShareHelper public immutable pricePerShareHelper;

     /// @dev config Unused members: chainlinkFeed, oracleError, oracleTimeout
-    /// @dev config.erc20 should be a RewardableERC20
+    /// @dev config.erc20 should be a Yearn V2 Vault for a fiatcoin curve pool
     constructor(
         CollateralConfig memory config,
         uint192 revenueHiding,
```

Incorrect gap sizes in multiple contracts

Multiple contracts with no descendants have incorrect gap sizes. Since these contracts do not have descendants yet, it is safe to normalize their gap sizes to 50 storage slots per contract. Below is a list of these contracts:

1. *RToken* – uses 14 slots and its gap size must be 16.
2. *AssetRegistry* – uses 6 slots and its gap size must be 44.
3. *Distributor* – uses 9 slots and its gap size must be 41.
4. *BasketHandler* – uses 21 slots and its gap size must be 29.

Use EMA Price Oracle without a wrapper

SFraxEthCollateral uses the *CurvePoolEmaPriceOracleWithMinMax*, which restricts prices to set bounds. There is no value in having prices restricted since *SFraxCollateral* will be marked as IFFY when it depegs.

Further, any incorrect **min** and **max** configuration that prevents a depeg from being recognized in the collateral plugin is a risk to the core protocol.

```
    function _getCurvePoolToken1EmaPrice() internal view returns (uint256
_token1Price) {
        _token1Price =
IEmaPriceOracleStableSwap(CURVE_POOL_EMA_PRICE_ORACLE).price_oracle();
    }
```

Unstable heartbeat in Frax Oracle

Based on FraxOracle docs, anyone can call the *addRoundData()* function on the Dual Oracle to add a fresh price to the Frax Oracle, they just need to pay gas.

Unlike Chainlink's stake-slash mechanism, there is no incentive for users to keep prices fresh in Frax Oracle unless it is officially maintained, and as a consequence the timeout check in *FraxOracleLib* may fail due to an unstable heartbeat.

There might be some other logic in *FraxOracleLib* that is not applicable to the Frax Oracle. For example, *latestRoundData()* does not revert if **priceSource() == address(0)** since it doesn't downstream rely on *priceSource().*

FraxOracleLib is not used currently. It is recommended not to use it in the future either and to remove it.

Flawed slippage control in DutchTradeRouter._placeBid()

DutchTradeRouter._placeBid() requires that **sellAmt >= expectedSellAmt**. **expectedSellAmt** is equal to *trade.lot()*, and **sellAmt** is the amount of tokens received to settle the trade.

```
        uint256 expectedSellAmt = trade.lot(); // {qSellToken}
        trade.bidWithCallback(new bytes(0));

        sellAmt = out.sellToken.balanceOf(address(this)) - sellAmt; // {qSellToken}
        require(sellAmt >= expectedSellAmt, "insufficient amount out");
```

However, *DutchTrade.settle()* sends the *lot()* amount of tokens to the **bidder**, i.e., **sellAmt** is always equal to **expectedSellAmt** (fee-on-transfer tokens and rebasing tokens are not supported by Reserve). If the protocol intends to use the check for slippage control it will not work.

```
        } else if (bidType == BidType.TRANSFER) {
            soldAmt = lot(); // {qSellTok}
            sell.safeTransfer(bidder, soldAmt); // {qSellTok}
        }
```

It is also worth noting that since *DutchTradeRouter* is end-user facing and **buyAmt** is affected by the price curve, it should be considered to provide a parameter for the user to control the slippage on **buyAmt**.

```
        out.buyAmt = trade.bidAmount(block.number); // {qBuyToken}
```

It is recommended to allow users to control slippage.

```
diff --git a/contracts/plugins/trading/DutchTradeRouter.sol
b/contracts/plugins/trading/DutchTradeRouter.sol
index acc1b091..7d08c76e 100644
--- a/contracts/plugins/trading/DutchTradeRouter.sol
+++ b/contracts/plugins/trading/DutchTradeRouter.sol
@@ -45,16 +45,23 @@ contract DutchTradeRouter is IDutchTradeCallee {
     );
     DutchTrade private _currentTrade;

+    uint256 private _maxBuyAmount;
+
     /// Place a bid on an OPEN dutch auction
     /// @param trade The DutchTrade to bid on
     /// @param recipient The recipient of the tokens out
     /// @dev Requires msg.sender has sufficient approval on the tokenIn with router
     /// @dev Requires msg.sender has sufficient balance on the tokenIn
     function bid(DutchTrade trade, address recipient) external returns (Bid memory) {
+        return bidWithSlippageProtection(trade, recipient, type(uint256).max);
+    }
+
+    function bidWithSlippageProtection(DutchTrade trade, address recipient, uint256
maxBuyAmount) public returns (Bid memory) {
+        _maxBuyAmount = maxBuyAmount;
         Bid memory out = _placeBid(trade, msg.sender);
         _sendBalanceTo(out.sellToken, recipient);
         _sendBalanceTo(out.buyToken, recipient);
         return out;
     }

     /// @notice Callback for DutchTrade
@@ -67,6 +74,7 @@ contract DutchTradeRouter is IDutchTradeCallee {
         bytes calldata
     ) external {
         require(msg.sender == address(_currentTrade), "Incorrect callee");
+        require(buyAmount < _maxBuyAmount);
         IERC20(buyToken).safeTransfer(msg.sender, buyAmount); //  {qBuyToken}
     }
```

Centralization risks

Centralized RSR supply

All RToken instances deployed via the Reserve *Deployer* are governed by staking the same RSR token. Currently, almost 50% of it is owned by Reserve.

The RSR that is owned by the Reserve Team can be accessed after a 28-day delay period.

If users are concerned about the RSR withdrawal, they can sell or redeem their rTokens within the 28-day period.

Still, there remains the risk for those that have deployed their RToken instance that it could be taken over by a compromise of the Reserve Team or any other entity with enough RSR.

Malicious Governance can steal all assets

The *Governance* has full control over the RToken instance it owns and is not bound by any constraints since the core logic contracts are upgradeable. Therefore, it depends on the specific RToken instance and the parameters of its *Governance* contract whether users have sufficient time to redeem their rTokens / unstake their RSR if they fear malicious behavior.

Furthermore, it is important to mention the special role of freezers and pausers that can freeze and pause certain functionality of the protocol. For example, if the *Governance* is malicious and is collaborating with a malicious freezer, rToken redemption and RSR unstaking can be frozen such that users cannot withdraw their assets before the *Governance* can enact a malicious proposal that upgrades the RToken instance and steals the users' assets.

The Reserve protocol is a complex system and the specific risks with regards to *Governance* depend on the individual RToken instance and its parameters.

Systemic risks

In StRSR, era changes and dynamic staking incentives can lead to unstable Governance

When RSR stakers are wiped out due to their RSR being seized to restore collateralization, a new era is entered. In addition, a new era can be entered manually if the **stakeRate** becomes unsafe. To enter a new era, the *Governance* must call *StRSR.resetStakes()*. The function documentation describes a standoff scenario whereby griefers can stake enough RSR to vote against the reset.

In addition, if *Governance* parameters are chosen unsafely, there may be insufficient time for users to stake again after an era change such that an attacker could more easily attack the *Governance* by staking a large amount of RSR.

More generally users are incentivized to stake RSR by receiving a share of the revenue that the RToken generates. If the economic incentives leave stakers better off staking in another RToken or selling their RSR, this makes the RToken vulnerable to takeovers.

RToken inherits risks of external collateral tokens

RToken instances are backed by a basket of collaterals. This means that the RToken derives its value from other assets and to assess the risk of the RToken, the risks of its underlying collateral tokens need to be considered.

For example, an RToken may only be backed by fully decentralized assets that can be considered safe such as WETH or WBTC. On the other hand, an RToken may be backed by assets that have many risks such as being controlled by a single entity, have an increased risk of being hacked or that are at risk of depegging. There is no guarantee that rToken can be redeemed for what the underlying collateral tokens represent if they're just on-chain representations of real-world assets.

Importantly, the damage that a single collateral can cause is limited to the value that the RToken holds in this collateral. This means that the RToken instance can in the worst case unregister the bad collateral and continue operation with the other collaterals.

Governance may become inactive

For proposals to succeed, a certain percentage of RSR stakers needs to cast their votes. This percentage is determined by the **quorumPercent** governance parameter.

If a lot of RSR stakers become inactive, proposals may not be able to succeed, such that the RToken is ungoverned which among other things means that the basket cannot be changed, and assets cannot be registered/unregistered.

To resolve this situation, new RSR stakers need to come in which may not occur depending on the incentives they have for staking their RSR.

Oracles must be trusted to report correct prices

Asset/collateral plugins use Chainlink oracles to report prices to the core logic contracts. The core contracts can handle situations when oracles stop working due to, for example timeouts or being deprecated.

However, if an oracle reports incorrect prices, this could lead to serious disruption and a loss to rToken holders as well as **rsr** stakers.

There are many paths how an incorrect price can lead to a loss depending on the specific plugin as different plugins make different use of oracles.

Governance proposals can be front-run

RToken instances can manage large amounts of assets and governance proposals can indirectly trigger trades, e.g., by changing the prime basket. Since governance proposals are visible to everyone it is possible to front-run them and to buy and sell assets before the governance proposal passes in anticipation of the price movement caused by the proposal.

As a result, the trades initiated by the proposal are executed at less favorable prices which leads to a loss for rToken holders, RSR stakers and other revenue destinations.

The more assets the RToken has under management and the less liquid these assets are, the higher the risk of such front-running.