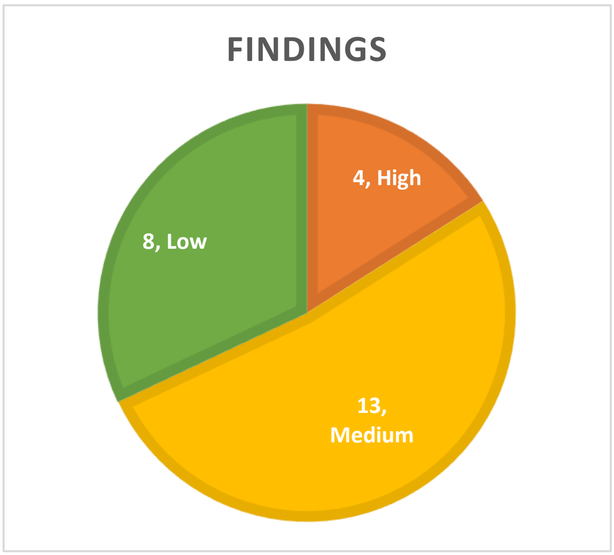
Trust Security

Smart Contract Audit

Reserve Protocol

02/01/24

Executive summary

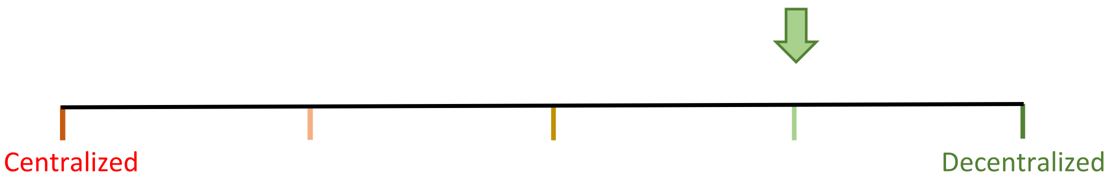


Category	Stablecoin
Audited file count	77
Lines of Code	7041
Auditor	cccz HollaDieWaldfee gjaldon
Time period	01/09/2023-09/10/2023

Findings

Severity	Total	Fixed	Open	Acknowledged
High	4	2	2	-
Medium	13	11	1	1
Low	8	7	-	1

Centralization score



Signature

Digitally signed by Trust
Date: 2024.01.02 11:50:27 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	09/10/2023	Client report
0.2	02/01/2024	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

- /contracts/plugins/assets/aave/StaticATokenLM.sol
- /contracts/p1/StRSR.sol
- /contracts/libraries/Fixed.sol
- /contracts/p1/BasketHandler.sol
- /contracts/plugins/assets/curve/PoolTokens.sol
- /contracts/p1/RToken.sol
- /contracts/p1/mixins/RecollateralizationLib.sol
- /contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol
- /contracts/p1/Broker.sol
- /contracts/p1/Deployer.sol
- /contracts/p1/StRSRVotes.sol
- /contracts/plugins/trading/DutchTrade.sol
- /contracts/p1/BackingManager.sol
- /contracts/plugins/governance/Governance.sol
- /contracts/p1/mixins/BasketLib.sol
- /contracts/p1/AssetRegistry.sol
- /contracts/plugins/trading/GnosisTrade.sol
- /contracts/p1/Distributor.sol
- /contracts/plugins/assets/FiatCollateral.sol
- /contracts/p1/RevenueTrader.sol
- /contracts/plugins/assets/compoundv3/WrappedERC20.sol
- /contracts/plugins/assets/RTokenAsset.sol
- /contracts/plugins/assets/Asset.sol
- /contracts/mixins/Auth.sol
- /contracts/p1/mixins/TradeLib.sol
- /contracts/plugins/assets/curve/CurveStableCollateral.sol
- /contracts/plugins/assets/curve/CurveStableMetapoolCollateral.sol
- /contracts/plugins/assets/erc20/RewardableERC20.sol
- /contracts/plugins/assets/aave/IStaticATokenLM.sol
- /contracts/mixins/ComponentRegistry.sol
- /contracts/plugins/assets/compoundv3/CTokenV3Collateral.sol
- /contracts/plugins/assets/stargate/StargatePoolFiatCollateral.sol
- /contracts/p1/mixins/Trading.sol
- /contracts/plugins/assets/morpho-aave/MorphoTokenisedDeposit.sol

- /contracts/plugins/assets/AppreciatingFiatCollateral.sol
- /contracts/plugins/assets/stargate/StargateRewardableWrapper.sol
- /contracts/plugins/assets/compoundv2/CTokenSelfReferentialCollateral.sol
- /contracts/libraries/Throttle.sol
- /contracts/plugins/assets/cbeth/CBETHCollateral.sol
- /contracts/p1/Furnace.sol
- /contracts/plugins/assets/ankr/AnkrStakedEthCollateral.sol
- /contracts/plugins/assets/lido/LidoStakedEthCollateral.sol
- /contracts/plugins/assets/morpho-aave/MorphoAaveV2TokenisedDeposit.sol
- /contracts/plugins/assets/erc20/RewardableERC20Wrapper.sol
- /contracts/plugins/assets/morpho-aave/IMorpho.sol
- /contracts/plugins/assets/morpho-aave/MorphoSelfReferentialCollateral.sol
- /contracts/plugins/assets/rocket-eth/RethCollateral.sol
- /contracts/p1/Main.sol
- /contracts/plugins/assets/OracleLib.sol
- /contracts/p1/mixins/Component.sol
- /contracts/plugins/assets/EURFiatCollateral.sol
- /contracts/plugins/assets/compoundv2/CTokenFiatCollateral.sol
- /contracts/plugins/assets/compoundv2/CTokenNonFiatCollateral.sol
- /contracts/plugins/assets/compoundv3/ICusdcV3Wrapper.sol
- /contracts/plugins/assets/compoundv3/CometHelpers.sol
- /contracts/plugins/assets/NonFiatCollateral.sol
- /contracts/plugins/assets/morpho-aave/MorphoNonFiatCollateral.sol
- /contracts/plugins/assets/curve/crv/CurveGaugeWrapper.sol
- /contracts/plugins/assets/frax-eth/SFraxEthCollateral.sol
- /contracts/plugins/assets/curve/CurveStableRTokenMetapoolCollateral.sol
- /contracts/plugins/assets/erc20/RewardableERC4626Vault.sol
- /contracts/libraries/Permit.sol
- /contracts/plugins/assets/dsr/SDaiCollateral.sol
- /contracts/plugins/assets/stargate/StargatePoolETHCollateral.sol
- /contracts/plugins/assets/SelfReferentialCollateral.sol
- /contracts/plugins/assets/morpho-aave/MorphoFiatCollateral.sol
- /contracts/plugins/assets/compoundv2/CTokenWrapper.sol
- /contracts/plugins/assets/aave/ATokenFiatCollateral.sol
- /contracts/libraries/Array.sol
- /contracts/p1/mixins/RewardableLib.sol
- /contracts/libraries/String.sol
- /contracts/libraries/NetworkConfigLib.sol
- /contracts/plugins/assets/compoundv2/ICToken.sol
- /contracts/plugins/assets/aave/StaticATokenErrors.sol
- /contracts/mixins/Versioned.sol
- /contracts/plugins/assets/VersionedAsset.sol
- /contracts/plugins/assets/compoundv3/IWrappedERC20.sol

Repository details

- **Repository URL:** https://github.com/reserve-protocol/protocol
- **Commit hash:** 722fea63e9fd30802b1278bc9763c9b0fed80d71
- **Mitigation review hash:** 3a4ac7c5f1f8cb269b2dfe941700fd5f41dc90ed

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 MorphoNonFiatCollateral is deployed with incorrect price feed

- **Category:** Configuration issues
- **Source:** deploy_morpho_aavev2_plugin.ts
- **Status:** Fixed

Description

The deployment script for *MorphoNonFiatCollateral* is configured to deploy using the wBTC/BTC feed as both the **chainlinkFeed** and the **targetUnitChainlinkFeed**.

```
const collateral = await NonFiatCollateralFactory.connect(deployer).deploy(
  {
    priceTimeout: priceTimeout,
    oracleError: combinedBTCWBTCError,
    maxTradeVolume: fp('1e6'), // $1m,
    oracleTimeout: oracleTimeout(chainId, '3600'), // 1 hr
    targetName: ethers.utils.formatBytes32String('BTC'),
    defaultThreshold: fp('0.01').add(combinedBTCWBTCError), // ~3.5%
    delayUntilDefault: bn('86400'), // 24h
    chainlinkFeed: networkConfig[chainId].chainlinkFeeds.WBTC!, (
    erc20: maWBTC.address,
  },
    revenueHiding,
    networkConfig[chainId].chainlinkFeeds.wBTCBTC!,
    oracleTimeout(chainId, '86400').toString() // 1 hr
)
```

In the deployment script above, **chainlinkFeed** is configured to use the **WBTC** feed while the **targetUnitChainlinkFeed** is configured to use the **wBTCBTC** feed. However, both feeds point to the same Chainlink feed address, which is the **wBTC/BTC** feed. Below is the configuration found at *common/configuration.ts*.

```
WBTC: '0xfdFD9C85aD200c506Cf9e21F1FD8dd01932FBB23',   (
BTC: '0xF4030086522a5bEEa4988F8cA5B36dbC97BeE88c',
EURT: '0x01D391A48f4F7339aC64CA2c83a07C22F95F587a',
EUR: '0xb49f677943BC038e9857d61E7d053CaA2C1734C1',
CVX: '0xd962fC30A72A84cE50161031391756Bf2876Af5D',
CRV: '0xCd627aA160A6fA45Eb793D19Ef54f5062F20f33f',
stETHETH: '0x86392dc19c0b719886221c78ab11eb8cf5c52812',
stETHUSD: '0xCfE54B5cD566aB89272946F602D76Ea879CAb4a8',
rETH: '0x536218f9E9Eb48863970252233c8F271f554C2d0',
cbETH: '0xf017fcb346a1885194689ba23eff2fe6fa5c483b',
wBTCBTC: '0xfdFD9C85aD200c506Cf9e21F1FD8dd01932FBB23', (
```

This misconfiguration leads to pricing the **maWBTC** collateral at the exchange rate of **wBTC** to **BTC**, which is close to 1. Since all oracle prices are in the Unit of Account, which is **USD**, the exchange rate of 1 means that 1 **wBTC** is assumed to be worth 1 **USD**.

If it were properly configured, the returned price would be closer to ~28,000 USD, which is the **USD** price of **wBTC** at the time of writing.

Recommended mitigation

The **chainlinkFeed** must be set to use the **wBTC/USD** feed, while the **targetUnitChainlinkFeed** must be set to use the **wBTC/BTC** feed.

Note that this issue is closely related to *M-10*.

Note further that there exist multiple issues in the deployment scripts that need to be addressed. They are discussed in the "Additional recommendations" section.

Team response

Fixed.

Mitigation Review

The fix corrects the configuration.

TRST-H-2 Attacker can steal rewards in MorphoAaveV2TokenisedDeposit

- **Category:** Logical flaws
- **Source:** MorphoAaveV2TokenisedDeposit.sol
- **Status:** Open

Description

According to the documentation, *MORPHO* rewards of *MorphoAaveV2TokenisedDeposit* can be sent to the contract by any account calling *RewardsDistributor.claim()*, and the rewards need to be synchronized with off-chain data before they can be claimed.

```
# Claiming rewards
Unfortunately Morpho uses a rewards scheme that requires the results of off-chain
computation to be piped into an on-chain function, which is not possible to do with
Reserve's collateral plugin interface. https://integration.morpho.xyz/track-and-manage-
position/manage-positions-on-morpho/claim-morpho-rewards claiming rewards for this
wrapper can be done by any account, and must be done on Morpho's rewards distributor
contract https://etherscan.io/address/0x3b14e5c73e0a56d607a8688098326fd4b4292135
```

However, *RewardableERC20* distributes the received rewards to the users who deposited before receiving the rewards depending only on the users' deposited amounts, not on how long they have been depositing.

A malicious user can perform **deposit assets -> call *RewardsDistributor.claim()* -> claim most of the *Morpho* rewards in *MorphoAaveV2TokenisedDeposit* -> withdraw assets** in a single transaction via a flash loan. This leads to attackers being able to steal almost all *MORPHO* rewards in *MorphoAaveV2TokenisedDeposit*.

MORPHO is currently non-transferable, and it is up to the DAO to enable the transfer function at the right time. However, this does not affect the issue, as the attacker can wait until the transfer function is enabled to collect the accumulated reward.

```
function    rewardTokenBalance(address    account)    external    returns    (uint256
claimableRewards) {
    _claimAndSyncRewards();
    _syncAccount(account);
    claimableRewards = accumulatedRewards[account] - claimedRewards[account];
}
```

Recommended mitigation

MORPHO protocol is aware of this issue, and *Morpho*'s official *ERC4626* vault sends the collected *MORPHO* rewards to a recipient address and then suggests redistribution using the script in *morpho-optimizers-rewards*. It is recommended to use the official script to distribute *MORPHO* rewards.

Team response

Fixed.

Mitigation Review

The issue has been mitigated with regards to the flash loan attack. However, rewards are still not distributed fairly. Users that have earned the rewards are not necessarily the ones that receive the rewards, while they can make deposits into the contract in anticipation of rewards that were earned in a previous period and will now be paid out.

Presumably, this is a known limitation and is favored over rewriting the entire contract logic to be able to pay out rewards fairly.

Another problem is that the **PAYOUT_PERIOD**, which is equal to **7 days**, is just the lower boundary for how long it takes to pay out all rewards.

The relevant code is displayed below:

```
uint256 amtToPayOut = state.pendingBalance * timeDelta / PAYOUT_PERIOD;
        state.pendingBalance -= amtToPayOut;
```

If **timeDelta = 7 days**, all rewards are paid out. But assume this function is called multiple times over that **7-day** period, say every **1 days**.

Paying out **1/7th** of the remaining rewards each day doesn't add up to **100%** of the rewards after **7 days** but rather it results in $1 - \left(\frac{6}{7}\right)^7 \approx 66\%$ being paid out after **7 days**.

Acknowledging the current limitations, the following minor recommendations can still be applied:

```
diff --git a/contracts/plugins/assets/morpho-aave/MorphoTokenisedDeposit.sol
b/contracts/plugins/assets/morpho-aave/MorphoTokenisedDeposit.sol
index 7785a987..23c41c1f 100644
--- a/contracts/plugins/assets/morpho-aave/MorphoTokenisedDeposit.sol
+++ b/contracts/plugins/assets/morpho-aave/MorphoTokenisedDeposit.sol
@@ -60,7 +60,7 @@ abstract contract MorphoTokenisedDeposit is RewardableERC4626Vault {
         if (timeDelta > PAYOUT_PERIOD) {
             timeDelta = PAYOUT_PERIOD;
         }
-        uint256 amtToPayOut = (state.pendingBalance * ((timeDelta * 1e18) /
PAYOUT_PERIOD)) / 1e18;
+        uint256 amtToPayOut = state.pendingBalance * timeDelta / PAYOUT_PERIOD;
         state.pendingBalance -= amtToPayOut;
         state.availableBalance += amtToPayOut;

@@ -78,8 +78,8 @@ abstract contract MorphoTokenisedDeposit is RewardableERC4626Vault {
    }

    function _distributeReward(address account, uint256 amt) internal override {
-       state.totalPaidOutBalance += uint256(amt);
-       state.availableBalance -= uint256(amt);
+       state.totalPaidOutBalance += amt;
+       state.availableBalance -= amt;
        SafeERC20.safeTransfer(rewardToken, account, amt);
    }
```

TRST-H-3 CusdcV3Wrapper._deposit() calls the incorrect hasPermission() function

- **Category:** Logical flaws
- **Source:** CusdcV3Wrapper.sol
- **Status:** Fixed

Description

CusdcV3Wrapper inherits from *WrappedERC20* and uses the *hasPermission()* function to manage approvals for wcUSDCv3.

For example, if Alice approves Bob to spend her wcUSDCv3, then Bob can arbitrarily transfer Alice's wcUSDCv3.

```
function approve(address spender, uint256 amount) public virtual override returns (bool)
{
    if (spender == address(0)) revert ZeroAddress();
    if (amount == type(uint256).max) {
        _allow(msg.sender, spender, true);
    } else if (amount == 0) {
        _allow(msg.sender, spender, false);
    } else {
        revert BadAmount();
    }
        return true;
    }
...
function transferFrom(
    address from,
    address to,
    uint256 amount
) public virtual override returns (bool) {
    if (!hasPermission(from, msg.sender)) revert Unauthorized();
    _transfer(from, to, amount);
    return true;
}
```

The problem is that in *CusdcV3Wrapper._deposit()*, as long as *hasPermission()* is true, then Bob can deposit Alice's cUSDCv3 instead of wcUSDCv3 to any address and thereby steal it (requires Alice to approve *CusdcV3Wrapper* to spend her cUSDCv3).

```
function _deposit(
    address operator,
    address src,
    address dst,
    uint256 amount
) internal {
    if (!hasPermission(src, operator)) revert Unauthorized();
    // {Comet}
    uint256 srcBal = underlyingComet.balanceOf(src);
    if (amount > srcBal) amount = srcBal;
    if (amount == 0) revert BadAmount();

    underlyingComet.accrueAccount(address(this));
    underlyingComet.accrueAccount(src);

    CometInterface.UserBasic              memory              wrappedBasic              =
underlyingComet.userBasic(address(this));
    int104 wrapperPrePrinc = wrappedBasic.principal;

    IERC20(address(underlyingComet)).safeTransferFrom(src, address(this), amount);
```

For example, Alice has 6000 cUSDCv3, she deposits 1000 cUSDCv3 for 1000 wcUSDCv3, and Alice calls *WrappedERC20.approve()* to allow Bob to spend her wcUSDCv3, then Bob should only be able to spend Alice's 1000 wcUSDCv3. However, this issue allows Bob to deposit Alice's remaining 5000 cUSDCv3 to any address. This should only be possible when Alice approves Bob to spend her cUSDCv3.

Recommended mitigation

It is recommended to use the u*nderlyingComet.hasPermission()* function here.

```
    function _deposit(
        address operator,
        address src,
        address dst,
        uint256 amount
    ) internal {
-       if (!hasPermission(src, operator)) revert Unauthorized();
+       if (!underlyingComet.hasPermission(src, operator)) revert Unauthorized();
```

Team response

Fixed.

Mitigation Review

The fix correctly calls *underlyingComet.hasPermission()* for permission checking.

TRST-H-4 Not all reward tokens are claimed in CurveGaugeWrapper

- **Category:** Logical flaws
- **Source:** CurveGaugeWrapper.sol
- **Status:** Open

Description

CurveGaugeWrapper inherits from *RewardableERC20* which claims rewards for users with *_claimAccountRewards()*. It claims rewards for only one token.

```
function _claimAccountRewards(address account) internal {
    uint256 claimableRewards = accumulatedRewards[account] - claimedRewards[account];

    emit RewardsClaimed(IERC20(address(rewardToken)), claimableRewards);

    if (claimableRewards == 0) {
        return;
    }

    claimedRewards[account] = accumulatedRewards[account];

    uint256 currentRewardTokenBalance = rewardToken.balanceOf(address(this));

    // This is just to handle the edge case where totalSupply() == 0 and there
    // are still reward tokens in the contract.
    uint256 nonDistributed = currentRewardTokenBalance > lastRewardBalance
        ? currentRewardTokenBalance - lastRewardBalance
        : 0;

    rewardToken.safeTransfer(account, claimableRewards);
```

Also, when *CurveGaugeWrapper* claims rewards for itself, it only claims CRV rewards, since *MINTER* is the CRV minter contract.

```
Function _claimAssetRewards() internal virtual override {
    MINTER.mint(address(gauge));
}
```

The assumption that all Curve Pools only have CRV rewards is incorrect. *LiquidityGaugeV2* to *V5* support multiple reward tokens beyond CRV and there's even a *RewardsOnlyGauge* that does not reward CRV at all.

Here's the code for *LiquidityGaugeV2* which allows for an array of reward tokens.

If a Curve Pool with multiple reward tokens is used as an underlying for *CurveGaugeWrapper*, all the other reward tokens that are not CRV will never be claimed and distributed to the wrapper's depositors. These unclaimed rewards will be lost forever since *CurveGaugeWrapper* has no functionality for claiming and distributing them.

The **MIM-THREE-POOL**, for example, has the SPELL token configured as its reward token. A *CurveGaugeWrapper* instance is already deployed that is using the **MIM_THREE_POOL** and it will not be able to collect any of the SPELL token rewards of its depositors.

Recommended mitigation

In *CurveGaugeWrapper._claimAssetRewards()*, there must be a configurable flag that can be set by the contract deployer so that the wrapper uses the correct rewards claiming function for its *LiquidityGauge*. For Curve's *LiquidityGauge* contract, *MINTER.mint()* needs to be called. For LiquidityGaugeV2-V5, *LiquidityGauge.claim_rewards()* needs to be called as well for the reward tokens to be claimed.

Note that each Curve Pool is configured differently and deployed with many different combinations of different versions of Pools, Gauges, and LP Tokens. There are cases when both *MINTER.mint()* and *LiquidityGauge.claim_rewards()* need to be called and cases where only one of them needs to be called, so it's necessary to be able to configure each of the calls separately.

Also, *RewardableERC20* needs the ability to handle multiple reward tokens.

Team response

Acknowledged.

Mitigation Review

A comment has been added to *CurveGaugeWrapper*, explaining that only **CRV** rewards will be claimed:

```
// Note: Only supports CRV rewards. If a Curve pool with multiple reward tokens is
// used, other reward tokens beyond CRV will never be claimed and distributed to
// depositors. These unclaimed rewards will be lost forever.
```

More broadly, the *CurveGaugeWrapper* is not compatible with all gauges (regardless of the **CRV** issue).

The *RewardsOnlyGauge* does not have a *user_checkpoint()* function that other gauges have which means the *MINTER.mint()* call reverts.

It was recognized in the initial report, explaining the need for a more dynamically configurable *CurveGaugeWrapper*:

```
// Note that each Curve Pool is configured differently and deployed with many
different combinations of different versions of Pools, Gauges, and LP Tokens. There
are cases when both MINTER.mint() and LiquidityGauge.claim_rewards() need to be called
and cases where only one of them needs to be called, so it's necessary to be able to
configure each of the calls separately.
```

The further recommendation is to add documentation to *CurveGaugeWrapper* that each wrapper deployment must be tested (beyond just checking the number of reward tokens). It would also be helpful to mention in the documentation that the wrapper does not support all gauges, specifically, the *RewardsOnlyGauge*.

Medium severity findings

TRST-M-1 Trades cannot be settled when distribution is changed to zero

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

Description

GnosisTrade and *DutchTrade* require their *settle()* function to be called to return profit / remaining sell balance to the **origin**.

The **origin** can be either *rsrTrader*, *rTokenTrader* or *BackingManager*.

The problem is with *rsrTrader* and *rTokenTrader* which both are instances of *RevenueTrader*.

RevenueTrader calls *ITrade.settle()* in its *settleTrade()* function which downstream relies on *Distributor.distribute()* to succeed.

However, *Distributor.distribute()* requires that **totalShares** is greater than zero.

A possible scenario is that the **governance** decides not to distribute either **rsr** or **rToken** at all which means the tokens should be returned to the *BackingManager* (at a time when there are ongoing trades).

But in this case the ongoing trades cannot be settled.

It's unintended that the traded amount MUST be distributed for settlement to occur.

In the case of *DutchTrade*, changing the distribution to zero also has the added consequence of not allowing users to bid on it.

This means that either the trade settles at a lower price when the distribution is changed to a non-zero value again or does not settle at all. So *DutchTrade* is somewhat protected against this scenario since the **governance** may just wait with setting the distribution to a non-zero value until the **endBlock** is reached to then return the sell tokens safely to the **origin**.

Overall, this issue leads to an allocation of funds that is unintended by the **governance**. If there are ongoing trades, the **governance** might have to wait for all of them to finish to make the adjustment. When one trade is finished however, another might be ongoing, so the impact is not limited to the funds in one trade.

Recommended mitigation

Add a check in *RevenueTrader.settleTrade()* or *RevenueTrader._distributeTokenToBuy()* to only distribute tokens if **totalShares** for the **tokenToBuy** is greater than zero.

The tokens can then manually be returned to the *BackingManager* via the *RevenueTrader.returnTokens()* function.

```
function settleTrade(IERC20 sell) public override(ITrading, TradingP1) returns (ITrade
trade) {
         trade = super.settleTrade(sell); // nonReentrant
-        _distributeTokenToBuy();
+        RevenueTotals memory revTotals = distributor.totals();
+         if ((tokenToBuy == rsr && revTotals.rsrTotal > 0) || (address(tokenToBuy) ==
address(rToken) && revTotals.rTokenTotal > 0)) {
+            _distributeTokenToBuy();
+        }
         // unlike BackingManager, do _not_ chain trades; b2b trades of the same token
are unlikely
     }
```

Optionally additional logic can be implemented to automatically send the *ITrade.sell()* and *ITrade.buy()* tokens to the *BackingManager* in the **else** case.

The check that **totalShares** for **tokenToBuy** is greater than zero should also be added to *RevenueTrader.manageTokens()* to not open unnecessary trades and incur slippage.

```
         uint256 len = erc20s.length;
         require(len > 0, "empty erc20s list");
         require(len == kinds.length, "length mismatch");
```

```
+        require((tokenToBuy == rsr && revTotals.rsrTotal > 0) || (address(tokenToBuy)
== address(rToken) && revTotals.rTokenTotal > 0), "zero distribution");

         // Calculate if the trade involves any RToken
         // Distribute tokenToBuy if supplied in ERC20s list
```

Team response

Fixed.

Mitigation Review

The fix uses try catch to catch the case where *_distributeTokenToBuy()* fails.

TRST-M-2 In settleTrade(), Revenue should only be distributed when notTradingPausedOrFrozen is true

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

Description

RevenueTrader.settleTrade() downstream calls *Distributor.distribute()*.

The **notTradingPausedOrFrozen** modifier has been removed from *Distributor.distribute()* saying that the check is performed upstream.

This is true except for the case when *RevenueTrader.settleTrade()* is the caller.

The impact is that funds can leave the protocol via distributions when the protocol is **frozen** or trading is **paused** which should not be possible.

Recommended mitigation

Only call *Distributor.distribute()* if **notTradingPausedOrFrozen**.

In case trading is **paused** or the protocol is **frozen**, revenue can be manually distributed later via *RevenueTrader.distributeTokenToBuy()*.

```
function settleTrade(IERC20 sell) public override(ITrading, TradingP1) returns (ITrade
trade) {
         trade = super.settleTrade(sell); // nonReentrant
-        _distributeTokenToBuy();
+        if (!main.tradingPausedOrFrozen()) _distributeTokenToBuy();
         // unlike BackingManager, do _not_ chain trades; b2b trades of the same token
are unlikely
     }
```

Team response

Fixed.

Mitigation Review

The fix replaces *_distributeTokenToBuy()* with *distributeTokenToBuy()* to apply the *notTradingPausedOrFrozen()* modifier.

TRST-M-3 Price is underestimated when trade is ongoing

- **Category:** Logical flaws
- **Source:** RTokenAsset.sol
- **Status:** Open

Description

The *RTokenAsset.price()* function makes use of the **basketRange** to calculate the price range for the RToken.

The **basketRange** is calculated via the *RecollateralizationLibP1.basketRange()* function with the context of the *BackingManager*.

It's wrong because the calculation does not account for the assets that are out for trading. The *RecollateralizationLibP1.basketRange()* function only accounts for the balances of the *BackingManager* and the **rsr** balance of *StRSR*.

Assume there's been a recent basket switch and a large trade is ongoing which trades collateral A for collateral B. All of the UoA that is locked up in the trade is not accounted for when the number of baskets that the *BackingManager* can hold is calculated. The amount of assets out for trading does not have an upper limit across *RToken* deployments so it's possible that *RTokenAsset.price()* can even reach zero when all assets are out for trading.

Recommended mitigation

The *BackingManager* should keep track of the assets that are out for trading, and they should be considered in the *RecollateralizationLibP1.basketRange()* function.

This won't have any implications for when the *BackingManager* uses the *RecollateralizationLibP1* library as this can only occur when **tradesOpen == 0**.

Team response

Fixed.

Mitigation Review

This fix adds **tokensOut** variable in *BackingManager* to track traded out assets, however the **__gap** is now incorrect. It has been decreased by two slots, but only one additional storage slot is used by the new **tokensOut** state variable. Instead of the gap size of 37, it should be 38.

```
diff --git a/contracts/p1/BackingManager.sol b/contracts/p1/BackingManager.sol
index 1065bcc2..4fa94fe7 100644
--- a/contracts/p1/BackingManager.sol
+++ b/contracts/p1/BackingManager.sol
@@ -351,5 +351,5 @@ contract BackingManagerP1 is TradingP1, IBackingManager {
      * variables without shifting down storage in the inheritance chain.
      * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
      */
-    uint256[37] private __gap;
+    uint256[38] private __gap;
 }
```

TRST-M-4 Call reward accounting functions after sending reward tokens

- **Category:** Logical flaws
- **Source:** BackingManager.sol, Distributor.sol
- **Status:** Fixed

Description

BackingManager.forwardRevenue() sends **rsr** to *StRSR*.

Distributor.distribute() sends **rsr** to *StRSR* and **rToken** to *Furnace*.

In all cases the tokens that are sent are treated as rewards.

After sending rewards to *StRSR* the *StRSR.payoutRewards()* function should be called and after sending rewards to *Furnace* the *Furnace.melt()* function should be called.

Failing to do so can lead to unintended reward distribution because the pending reward balance is not updated, and it takes until the next reward distribution for the additional rewards to be registered.

Compare scenario 1 (no call to *Furnace.melt()* after sending rewards) with scenario 2 (call to *Furnace.melt()* after sending rewards):

```
Let's consider two periods with meltRatio = 0.1

Scenario 1:
At time 0, lastPayoutBal = 100.
At time 0 + PERIOD, 100 RToken is distributed to Furnace and no action is taken.
At time 0 + PERIOD * 2, the number of RToken melted is (1 - (1 - 0.1) ** 2) * 100 = 19.
The total RToken melted are 19.

Scenario 2:
At time 0, lastPayoutBal = 100.
At time 0 + PERIOD, 100 RToken is distributed to Furnace, melt() is called, the number
of RToken melted is (1 - (1 - 0.1) ** 1 ) * 100 = 10, and lastPayoutBal = 100 + 100 -
10 = 190.
At time 0 + PERIOD * 2, the number of RToken melted is (1 - (1 - 0.1) ** 1) * 190 = 19.
The total RToken melted are 29.
```

Recommended mitigation

Call the missing reward accounting functions to register the additional rewards after they are sent. In the case when *Distributor.distribute()* calls *Furnace.melt()*, this must be wrapped in a try-catch block as *Furnace.melt()* has the **notFrozen** modifier and *Distributor.distribute()* should be callable any time.

Team response

Fixed.

Mitigation Review

Furnace.melt() is allowed to be called when frozen, so the fix is fine.

TRST-M-5 Governance: Same-era check can be bypassed

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

Description

The *Governance* contract adds an additional access control around the *_execute()* function which requires that a proposal must be started in the same era that it is executed in.

The problem is that this check can be bypassed due to a misconfiguration in the *FacadeWrite* contract.

It grants the **EXECUTOR_ROLE** to **address(0)** which means anyone can execute a queued proposal directly via the *TimelockController* contract on a path that circumvents the same-era check.

This issue is a fundamental flaw in the *Governance* contract configuration. The ability to execute proposals when it shouldn't be allowed allows for a "High" impact to be possible. However, we estimate the likelihood of exploitation to be "Low". The proposal must have received sufficient votes to be queued in the previous era and the **guardian** could cancel the queued proposal (even though it must be assumed that the **guardian** wouldn't do this if this issue was exploited as a zero-day).

Recommended mitigation

Only grant the **EXECUTOR_ROLE** to the *Governance* contract which ensures proposals can only be executed after passing the same-era check.

Team response

Fixed.

Mitigation Review

The fix grants the **EXECUTOR_ROLE** to the *Governance.*

TRST-M-6 Minting revenue RToken dilutes backingBuffer in forwardRevenue()

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

Description

BackingManager.forwardRevenue() mints **rToken** for excess collateral but does not keep the **backingBuffer**.

```
uint192 needed = rToken.basketsNeeded().mul(FIX_ONE + backingBuffer); // {BU}
if (basketsHeld.bottom > needed) {
    rToken.mint(basketsHeld.bottom - needed);
    needed = rToken.basketsNeeded().mul(FIX_ONE + backingBuffer); // keep buffer
}
```

Consider the following scenario:

```
Given:
backingBuffer = 5%
rToken supply = 1000
basketsNeeded = 1000
basketsHeld.bottom = 1100
bu <-> rToken exchange rate: 1

There are:
needed = 1000 * 1.05 = 1050
rToken.mint(1100 - 1050) increases RToken supply by 50 RToken

The 5% buffer is not maintained. The buffer is now:
basketsHeld.bottom / basketsNeeded' - 1 = 1100 / 1050 - 1 = 4.7%
```

This shows that when revenue **rToken** is minted, **backingBuffer** is not kept at its full value, which makes the *RToken* more prone to becoming undercollateralized. In other words, the *BackingManager* allows more revenue to leave than it should.

Recommended mitigation

To ensure that **backingBuffer** is kept when revenue **rToken** is minted, perform the following calculation to determine the amount of baskets to mint:

```
Let x be the number of baskets to mint.

The following equation must be solved for x:
1 + backingBuffer = basketsHeld.bottom / (basketsNeeded + x)
<=> x = (basketsHeld.bottom / (1 + backingBuffer)) - basketsNeeded.

In the above example, x = (1100 / (1 + 5%)) - 1000 = 47.6.
Minting this number of baskets keeps the backingBuffer: 1100 / 1047.6 - 1 = 5%.
```

Team response

Fixed.

Mitigation Review

The fix modifies the algorithm to keep **backingBuffer**.

TRST-M-7 RToken should be refreshed

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

Description

For assets where *price()* may return stale prices, such as *sDAI* and *CToken*, the protocol will make the necessary function calls in *refresh()* to update the asset state, such as calling *drip()* in *SDaiCollateral.refresh()* and calling *exchangeRateCurrent()* in *CTokenFiatCollateral.refresh()*.

Therefore *RTokenAsset.refresh()* should also update the state of the *RToken* and try to ensure that calling *price()* returns the most recent price.

Calling *assetRegistry.refresh()* in *refresh()* ensures that *basketHandler.price()* in *tryPrice()* uses the fresh *RToken* collateral prices.

Calling *Furnace.melt()* ensures that *tryPrice()* uses the fresh *RToken.totalSupply()*.

```
function tryPrice() external view virtual returns (uint192 low, uint192 high) {
    (uint192 lowBUPrice, uint192 highBUPrice) = basketHandler.price(); // {UoA/BU}
    require(lowBUPrice != 0 && highBUPrice != FIX_MAX, "invalid price");
    assert(lowBUPrice <= highBUPrice); // not obviously true just by inspection

    // Here we take advantage of the fact that we know RToken has 18 decimals
    // to convert between uint256 an uint192. Fits due to assumed max totalSupply.
    uint192 supply = _safeWrap(IRToken(address(erc20)).totalSupply());

    if (supply == 0) return (lowBUPrice, highBUPrice);

    // The RToken's price is not symmetric like other assets!
    // range.bottom is lower because of the slippage from the shortfall
    BasketRange memory range = basketRange(); // {BU}

    // {UoA/tok} = {BU} * {UoA/BU} / {tok}
    low = range.bottom.mulDiv(lowBUPrice, supply, FLOOR);
    high = range.top.mulDiv(highBUPrice, supply, CEIL);

    assert(low <= high); // not obviously true
}
```

Recommended mitigation

It is recommended to change *RTokenAsset.refresh()* as follows:

```
    function refresh() public virtual override {
       // No need to save lastPrice; can piggyback off the backing collateral's saved
prices
+      if (msg.sender != address(assetRegistry) {
+          assetRegistry.refresh()
+      }
+      main.furnace.melt()
       cachedOracleData.cachedAtTime = 0; // force oracle refresh
    }
```

Note that it must be checked whether the caller is the **assetRegistry** as otherwise this would result in an infinite loop.

In addition, when *RTokenAsset* calls *RTokenAsset._updateCachedPrice()* to cache the most recent price, *refresh()* should be called before it.

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

TRST-M-8 Incorrect reward distribution in RewardableERC20

- **Category:** Logical flaws
- **Source:** RewardableERC20.sol

- **Status:** Fixed

Description

RewardableERC20._claimAndSyncRewards() retains some of the rewards due to rounding until the next distribution. However, this will cause the remaining rewards to be distributed to other users, thereby causing losses to the previous users.

```
function _claimAndSyncRewards() internal virtual {
     uint256 _totalSupply = totalSupply();
     if (_totalSupply == 0) {
         return;
     }
     _claimAssetRewards();
     uint256 balanceAfterClaimingRewards = rewardToken.balanceOf(address(this));

     uint256 _rewardsPerShare = rewardsPerShare;
     uint256 _previousBalance = lastRewardBalance;

     if (balanceAfterClaimingRewards > _previousBalance) {
         uint256 delta = balanceAfterClaimingRewards - _previousBalance;
         uint256 deltaPerShare = (delta * one) / _totalSupply;
         balanceAfterClaimingRewards    =    _previousBalance    +    (deltaPerShare    *
_totalSupply) / one;

         // {qRewards/share} += {qRewards} * {qShare/share} / {qShare}
         _rewardsPerShare += deltaPerShare;
     }

     lastRewardBalance = balanceAfterClaimingRewards;
     rewardsPerShare = _rewardsPerShare;
}
```

Consider the following scenario:

```
_decimals = 18

Alice deposits 1e6 * 1e18 tokens and receives 1e6 * 1e18 shares

After some time, Alice's reward is 1.7e6

Bob  deposits  1e6  *  1e18  tokens,  _claimAndSyncRewards  is  called  and  deltaPerShare  =
(1.7e6 * 1e18)/(1e6 * 1e18) = 1 (Round 1.7 to 1)

Therefore Alice's reward is 1 * 1e6 * 1e18 / 1e18 = 1e6

After another period of time, Alice and Bob increase their rewards by 1.7e6 * 2 = 3.4e6

Bob calls _claimAndSyncRewards and deltaPerShare = (0.7e6 + 3.4e6) * 1e18/(2e6 * 1e18)
= 2 (Round 2.05 to 2).
Bob's total reward is 2 * 1e6 * 1e18 / 1e18 = 2e6, but in fact Bob's reward should only
be 1.7e6. The contract distributes some of Alice’s reward to Bob.
```

According to the calculation, for a total supply of **N * 1e18** (or other decimals), there is a loss of at most **N-1** wei rewards.

For example, when USDC (6 decimals) is used as the staking token and the reward token, when 1e6 USDC is staked, there will be a maximum loss of nearly 1 USDC in reward tokens.

In the extreme case, when SHIB is used as the stake token and WBTC (8 decimals) is used as the reward token, when 1e8 SHIB is staked (worth 700 USD), there will be a maximum loss of nearly 1 WBTC in reward tokens.

Recommended mitigation

The recommendation is to use a larger multiplier to avoid rounding losses.

```
    constructor(IERC20 _rewardToken, uint8 _decimals) {
        rewardToken = _rewardToken;
-       one = 10**_decimals; // set via pass-in to prevent inheritance issues
+       one = 10**(_decimals+9); // set via pass-in to prevent inheritance issues
    }
```

Team response

Fixed.

Mitigation Review

The fix uses a larger multiplier to avoid rounding losses.

TRST-M-9 pegPrice is always 1

- **Category:** Logical flaws
- **Source:** SFraxEthCollateral.sol
- **Status:** Acknowledged

Description

SFraxEthCollateral.tryPrice() always returns the **pegPrice** with *targetPerRef()*.

```
function tryPrice()
    external
    view
    override
    returns (
        uint192 low,
        uint192 high,
        uint192 pegPrice
    )
{
    // {UoA/tok} = {UoA/target} * {ref/tok} * {target/ref} (1)
    uint192 p = chainlinkFeed.price(oracleTimeout).mul(_underlyingRefPerTok());
    uint192 err = p.mul(oracleError, CEIL);

    low = p - err;
    high = p + err;
    // assert(low <= high); obviously true just by inspection

    pegPrice = targetPerRef();
}
```

targetPerRef() is **FIX_ONE** which means a depeg cannot be detected.

```
function targetPerRef() public view virtual returns (uint192) {
    return FIX_ONE;
}
```

frxETH cannot be redeemed for ETH via the Frax protocol, so it is necessary to check whether frxETH (ref) has depegged from ETH (target).

Recommended mitigation

The team says they are aware of this issue.

It is recommended to at least document this issue, or implement an oracle (if available) to get the price of **frxETH/ETH**.

Team response

Acknowledged.

TRST-M-10 MorphoNonFiatCollateral works like self-referential collaterals

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

Description

MorphoNonFiatCollateral is meant to work like *CTokenNonFiatCollateral*. Instead, it works a lot like the CToken and Morpho self-referential collateral contracts. The non-fiat versions of Morpho and CToken were purposely built for non-fiat underlying assets that do not have price feeds in USD, which is the Unit of Account in Reserve. The main example for this is wBTC, which has no wBTC/USD Chainlink price feed in Ethereum. To get the USD price for wBTC with Chainlink, it would need both wBTC/BTC and BTC/USD Chainlink feeds.

Below is the *MorphoNonFiatCollateral.tryPrice()* function. Note that it works with two Chainlink feeds. It expects **chainlinkFeed** to be a {UoA/ref} feed. In the case of wBTC as reference token, **chainlinkFeed** should be the wBTC/USD feed. As stated earlier though, there is no wBTC/USD Chainlink feed on Ethereum. If there was, then *MorphoSelfReferential* should be used instead of *MorphoNonFiatCollateral*.

```
function tryPrice()
    external
    view
    override
    returns (
        uint192 low,
        uint192 high,
        uint192 pegPrice
    )
{
    pegPrice = targetUnitChainlinkFeed.price(targetUnitOracleTimeout); (

    // {UoA/tok} = {UoA/ref} * {ref/tok}
    uint192 p = chainlinkFeed.price(oracleTimeout).mul(_underlyingRefPerTok()); (
    uint192 err = p.mul(oracleError, CEIL);

    high = p + err;
    low = p - err;
}
```

Another issue is that **pegPrice** is not multiplied by the price from **chainlinkFeed**. The feedsa re also named incorrectly. The *CTokenNonFiatCollateral.tryPrice()* function is a good reference for what a correctly implemented non-fiat collateral contract looks like.

```
function tryPrice()
    external
    view
    override
    returns (
        uint192 low,
        uint192 high,
        uint192 pegPrice
    )
{
    pegPrice = chainlinkFeed.price(oracleTimeout); (

    // {UoA/tok} = {UoA/target} * {target/ref} * {ref/tok}
    uint192 p =
        targetUnitChainlinkFeed.price(targetUnitOracleTimeout).mul(pegPrice).mul(
            _underlyingRefPerTok()
        ); (
    uint192 err = p.mul(oracleError, CEIL);

    high = p + err;
    low = p - err;
}
```

Note that in the *CTokenNonFiatCollateral.tryPrice()* function, **pegPrice** is multiplied by the price returned by the **targetUnitChainlinkFeed**. Also, **chainlinkFeed** is used for the **pegPrice**.

Recommended mitigation

Modify the *MorphoNonFiatCollateral.tryPrice()* function to work just like the *CTokenNonFiatCollateral.tryPrice()* function and update all related deployment scripts accordingly.

Team response

Fixed.

Mitigation Review

MorphoNonFiatCollateral now mostly mirrors the implementation of *CTokenNonFiatCollateral*. There is, however, a minor difference in their implementation where *CTokenNonFiatCollateral* uses **chainlinkFeed** for its **pegPrice** like so:

```
pegPrice = chainlinkFeed.price(oracleTimeout);
```

In contrast to *MorphoNonFiatCollateral* which uses **targetUnitChainlinkFeed** for its **pegPrice**:

```
pegPrice = targetUnitChainlinkFeed.price(targetUnitOracleTimeout);
```

CTokenNonFiatCollateral uses incorrect monetary units for its variable names.

It is recommended to update *MorphoNonFiatCollateral* so that it uses **chainlinkFeed** for its **pegPrice** calculation and **targetUnitChainlinkFeed** for its **p** calculation instead of **pegPrice**. Also, switch the assignment in the deployment scripts.

That way, variable names and monetary units are consistent across the non-fiat collaterals.

TRST-M-11 CurveStableMetapoolCollateral should override refresh() function

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

Description

CurveStableRTokenMetapoolCollateral is a Metapool whose **pairedToken** is RToken. Due to the differences from *CurveStableMetapoolCollateral*, additional adaptation is required.

For example, *CurveStableRTokenMetapoolCollateral.refresh()* does not call *RToken.refresh()*. If *RToken.refresh()* involves some state updates, not calling it will result in using RToken's stale data, such as price. As mentioned in *M-07*, *RToken.price()* is likely to return a stale price.

Recommended mitigation

It is recommended to implement the *CurveStableRTokenMetapoolCollateral.refresh()* function as follows:

```
function refresh() public virtual override {
    pairedAssetRegistry.toAsset(pairedToken).refresh();
    super.refresh(); // already handles all necessary default checks
}
```

Team response

Fixed.

Mitigation Review

The fix implements *CurveStableRTokenMetapoolCollateral.refresh()* and calls *pairedAssetRegistry.refresh()* in it.

TRST-M-12 Use the most recent contract from Convex repository

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

Description

The *ConvexStakingWrapper* that's currently used in the Reserve repository (and which is not formally in scope for this audit) is from a previous commit in the Convex repository.

Based upon communication with the Reserve Team it was determined not to be necessary to use the *ConvexStakingWrapper* from the most recent commit.

However, there are some changes which need to be applied. For pools with an ID **>= 151**, the extra reward tokens are wrapped. This means there's an additional step necessary to get the token contract.

Currently, only the **eUSD_FRAX_BP** pool is affected by this as its ID is 156. Only by chance does this not lead to an issue as its extra token is **wrappedCVX** and failing to unwrap it has no effect since **CVX** is added as a reward token by default. With other curve pools this could lead to a loss of rewards.

Another notable change is that reward integrals are now stored in **uint256** as opposed to **uint128** variables as apparently storing them in **uint128** variables could have led to overflows and wrong reward accounting in some edge cases.

Recommended mitigation

The recommendation is to use the latest *ConvexStakingWrapper* contract to ensure compatibility with all pools and to apply the latest fixes.

We haven't followed up on the impact of every single fix that has been applied since based on the above description it is clear that the most recent version needs to be used.

Based on our cursory review of the commits we conclude that there is no risk for the currently deployed collateral plugins that use the old *ConvexStakingWrapper*. Nonetheless it is recommended to reach out to the Convex Team to assess whether assets are at risk and if the already deployed plugins can continue to be used.

Team response

Fixed.

Mitigation Review

The fix upgraded ConvexStakingWrapper to the latest version and adapted it.

TRST-M-13 warmupPeriod introduces DOS attack vector via oracle timeouts

- **Category:** Logical flaws
- **Source:** OracleLib.sol/BasketHandler.sol
- **Status:** Fixed

Description

When the **timeout** of an oracle is exceeded, the *OracleLib.price()* function reverts. This then causes the collateral plugin to mark the collateral as **IFFY**.

During the time a collateral in the basket is **IFFY**, certain actions cannot be performed, including rToken issuance, rsr withdrawals, rebalancing, revenue forwarding.

And as the basket then becomes **SOUND** again, a **warmupPeriod** needs to pass before the above actions are possible again.

The problem is that oracle timeouts are currently configured to be equal to the heartbeat time of the oracles.

For example, the DAI/USD oracle has a heartbeat time of 1 hour (3600 seconds) and the oracle timeout is also exactly 3600 seconds. Observing the on-chain time difference between price updates shows however that the difference can be slightly bigger than 3600 seconds. This means that there can be blocks in between price updates for which the last price update is more than 3600 seconds ago, meaning an attacker can call *AssetRegistry.refresh()* to DOS the above-mentioned actions.

```
Case study for the DAI/USD feed:

(https://etherscan.io/address/0xaed0c38402a5d19df6e4c03f4e2dced6e29c1ee9):

Heartbeat time is 3600 seconds and Reserve collaterals become IFFY for prices older than
3600 seconds.


roundId: 110680464442257314054

updatedAt: 1696498355


roundId: 110680464442257314055

updatedAt: 1696501991

block.number: 18283738


The new price for round 110680464442257314055 was observed after 3636 seconds.

In between lie blocks 18283737 and 18283736 that have a most recent price older than
3600 seconds.
```

Depending on the **warmupPeriod** the DOS can be permanent by consistently refreshing it. In this case it can only be resolved by the Governance switching collateral plugins.

Recommended mitigation

Determine a suitable buffer based on historical price updates that's added to the heartbeat time to arrive at the oracle timeout (a value of say 5 minutes, i.e., 300 seconds, seems appropriate).

Team response

Fixed.

Mitigation Review

In the initial report it has not been recognized that there existed in fact a **60 second** delay in the deployment scripts. As a result, the scenario described in the report with a delay of **36 seconds** could not have caused a DoS. Analysis of the Chainlink feeds showed that there is roughly a **1 percent** chance of updates taking longer than **60 seconds** when aggregating data for different feeds. In all the cases where this **60 second** delay wasn't enough, a **120 second** delay would have been enough.

Therefore, the **5 minutes** delay is more than sufficient to mitigate the DoS risk and prevent false positives from interrupting the operation of the protocol.

Low severity findings

TRST-L-1 Incorrect calculation of gap size

- **Category:** Upgradeability issues

- **Source:** Broker.sol
- **Status:** Fixed

Description

Broker incorrectly calculates the gap size. As part of the 3.1.0 release, the **rToken** variable has been introduced without reducing the gap size of the contract.

Recommended mitigation

The gap size should be reduced from **42** to **41** to account for the additional variable.

Team response

Fixed.

Mitigation Review

The fix reduces the gap size to 41.

TRST-L-2 Slippage should increase with auction volume

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

Description

The **worstPrice** that a *DutchTrade* can settle at is calculated from the **sellLow** and **buyHigh** price.

In addition to that it accounts for a **slippage** percentage that depends on the volume of the auction.

slippage decreases linearly from **100% * maxTradeSlippage** at **minTradeVolume** to **0% * maxTradeSlippage** at **maxTradeVolume**.

The purpose is to account for gas costs that make up a higher percentage of the auction volume at lower auction volumes and a lower percentage at higher auction volumes.

What's unaccounted for is the volume <-> liquidity relationship. There is lower liquidity at higher volumes. Failing to account for the volume <-> liquidity relationship prevents trades from settling at the lower end of the price range which should be acceptable (especially for very high volumes).

This can lead to repeatedly failing trades and delays in all components that rely on trades being settled. It introduces a small risk of DoS as there can only ever be one trade at a time in the *BackingManager* and one trade per ERC20 in the *RevenueTrader*.

Recommended mitigation

For all auction volumes the **maxTradeSlippage** should be used to calculate **worstPrice**.

This means that the **slippage** increases in terms of UoA as auction volume increases which accounts for the volume <-> liqudity relationship in addition to the gas costs.

Team response

Fixed.

Mitigation Review

This fix implements the recommendation.

TRST-L-3 Incorrect calculation of endTime

- **Category:** Off-by-one errors
- **Source:** DutchTrade.sol
- **Status:** Fixed

Description

The **endTime** variable is incorrectly calculated in the *DutchTrade.init()* function. It is off by one block.

The variable does not have internal security implications and only needs to be viewed externally.

Recommended mitigation

Calculate **endTime** like this:

```
endTime = uint48(block.timestamp + ONE_BLOCK * (_endBlock - _startBlock + 1));
```

Team response

Fixed.

Mitigation Review

The fix changed the **endTime** as recommended.

TRST-L-4 stgETH triggers ETH callback

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

Description

The protocol does not support tokens with callback mechanisms. This extends to LP tokens with assets that have callback mechanisms such as the **stETH-ETH** Pool in Curve. This is a security measure to prevent reentrancy attacks.

StargatePoolETHCollateral however is a one-sided pool that supports **stgETH**. **stgETH** is Stargate's **ETH** wrapper token forked from **WETH9**. Its main difference is that it automatically unwraps **stgETH** to **ETH** on transfers. This leads to triggering a callback for any contract recipient and opens the possibility of reentrancy attacks.

We have done a cursory review of Stargate to check for reentrancy attacks but have not found any.

A more in-depth review would be necessary if the protocol would like to support Stargate's **ETH** pools.

Recommended mitigation

Either drop support of **stgETH** and remove *StargatePoolETHCollateral* or have an in-depth review done on Stargate and note in the documentation that the protocol is making an exception for Stargate's **ETH** Pool.

Team response

Fixed.

Mitigation Review

The fix removes the deployment script for *StargatePoolETHCollateral.*

TRST-L-5 Curve Metapools with Separate LP Tokens are unsupported

- **Category:** Logical flaws
- **Source:** CurveStableMetapoolCollateral.sol
- **Status:** Acknowledged

Description

CurveStableMetapoolCollateral expects that the Metapool and the Metapool's LP Token are one contract. This is not the case for all Metapools since some of the older Metapools found here have the LP Token contract separate from the Metapool contract.

This leads to calls to *metapoolToken.totalSupply()* and *metapoolToken.decimals()* in *CurveStableMetapoolCollateral.tryPrice()* to revert. It breaks compatibility and makes it impossible to register and use it in the system. This means that some Curve Metapools are unusable as collateral.

Recommended mitigation

Have separate immutables in *CurveStableMetapoolCollateral* for **metapool** and **metapoolToken**.

metapool would be the address for the Metapool contract and **metapoolToken** would be the address for the LP Token contract. All calls to Pool functions in *CurveStableMetapoolCollateral* should be replaced with **metapool** such as:

- *metapoolToken.coins() -> metapool.coins()*
- *metapoolToken.get_virtual_price() -> metapool.get_virtual_price()*
- *metapoolToken.balances() -> metapool.balances()*

Doing this allows Reserve to support Metapools with one contract for LP Token and Pool and Metapools with separate contracts for LP Token and Pool. For Metapools with one contract as LP Token and Pool, we use the same address for both **metapoolToken** and **metapool**.

Team response

Acknowledged.

TRST-L-6 Suboptimal reward claiming if allocation for pool is changed to zero

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

Description

The *StargateRewardableWrapper._claimAssetRewards()* function is used to claim rewards from the Stargate LPStaking contract:

```
function _claimAssetRewards() internal override {
    IStargateLPStaking.PoolInfo memory poolInfo = stakingContract.poolInfo(poolId);

    if (poolInfo.allocPoint != 0 && totalSupply() != 0) {
        stakingContract.deposit(poolId, 0);
    } else {
        stakingContract.emergencyWithdraw(poolId);
    }
}
```

The purpose of the **poolInfo.allocPoint != 0** check is to prevent a division-by-zero case in the *LPStaking* contract.

The division-by-zero error is caused when **totalAllocPoint = 0**, not when **poolInfo.allocPoint = 0** and therefore when **totalAllocPoint != 0** and **poolInfo.allocPoint = 0**, not all rewards will be claimed.

Recommended mitigation

To mitigate this issue, the **poolInfo.allocPoint != 0** check should be replaced with **stakingContract.totalAllocPoint() != 0**.

Furthermore, the **totalSupply != 0** check can be removed. If **totalSupply() == 0** it does no harm to call *deposit()* in case someone has sent LP tokens to the wrapper without depositing them. The LP tokens can remain staked to accrue rewards. New LP token depositors would thereby be slightly better off.

The **else** case can be removed entirely since emergency withdrawals can always be made and the *StargateRewardableWrapper._beforeWithdraw()* function will take care of withdrawing the LP tokens.

As a result, the recommended change is this:

```
function _claimAssetRewards() internal override {
        IStargateLPStaking.PoolInfo memory poolInfo = stakingContract.poolInfo(poolId);

-       if (poolInfo.allocPoint != 0 && totalSupply() != 0) {
+      if (stakingContract.totalAllocPoint() != 0) {
            stakingContract.deposit(poolId, 0);
-       } else {
-           stakingContract.emergencyWithdraw(poolId);
        }
```

```
    }
```

Team response

Fixed.

Mitigation Review

This fix implements the recommendation.

TRST-L-7 Restriction on Broker.reportViolation() prevents bids from being made

- **Category:** Logical flaws
- **Source:** DutchTrade.sol, Broker.sol
- **Status:** Fixed

Description

When a bid is made on a *DutchTrade* in the geometric decay phase, a violation is reported to the *Broker* via the *Broker.reportViolation()* function.

The problem is that the *Broker.reportViolation()* function has the **notTradingPausedOrFrozen** modifier which means the trade might have to settle at a lower price than possible when trading is **paused** or the protocol is **frozen**.

In addition, the violation would not be reported, and another trade could be opened, encountering the same pricing issue.

Recommended mitigation

It should be possible for *DutchTrade* to report a violation even when trading is **paused**, or the protocol is **frozen**. The recommendation therefore is to remove the **notTradingPausedOrFrozen** modifier from the *Broker.reportViolation()* function.

```
-   function reportViolation() external notTradingPausedOrFrozen {
+   function reportViolation() external {
        require(trades[_msgSender()], "unrecognized trade contract");
        ITrade trade = ITrade(_msgSender());
        TradeKind kind = trade.KIND();
```

Team response

Fixed.

Mitigation Review

The fix removes the **notTradingPausedOrFrozen** modifier from *Broker.reportViolation().*

TRST-L-8 Rewards may not be claimable due to rounding

- **Category:** Logical flaws
- **Source:** CusdcV3Wrapper.sol
- **Status:** Fixed

Description

CusdcV3Wrapper calculates the rewards that an **account** has accrued in the same way that *Comet* does for *CusdcV3Wrapper*.

The problem is that the rewards for *CusdcV3Wrapper* may be accrued more often (i.e., rounded down more often) than the rewards for an individual **account** in *CusdcV3Wrapper*. This is easy to see in the extreme case when there is only a single **account** that has deposited into CusdcV3*Wrapper*.

In this case the rewards that CusdcV3Wrapper has accrued from *Comet* are less than the rewards that the **account** is supposed to receive from CusdcV3Wrapper by some small dust amount. This means that the *CusdcV3Wrapper.claimTo()* function reverts when **account** tries to claim its rewards.

This issue is unlikely as there are probably multiple accounts in the *CusdcV3Wrapper* and so rounding would favor the *Wrapper* as opposed to the individual accounts. Also, the revert can be prevented by sending a dust amount of **COMP**.

Recommended mitigation

It is recommended to limit the reward amount for an **account** in the *CusdcV3Wrapper.claimTo()* function to the **COMP** balance of the *Wrapper*.

```
        if (accrued > claimed) {
            owed = accrued - claimed;
            rewardsClaimed[src] = accrued;
            rewardsAddr.claimTo(address(underlyingComet),                address(this),
address(this), true);
+           uint256 bal = IERC20(rewardERC20).balanceOf(address(this));
+           if (owed > bal) owed = bal;
            IERC20(rewardERC20).safeTransfer(dst, owed);
        }
```

Team response

Fixed.

Mitigation Review

The cast to *IERC20* interface is redundant. Below is a recommended snippet.

```
diff --git a/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol
b/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol
index e07b30d5..101a2f76 100644
--- a/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol
+++ b/contracts/plugins/assets/compoundv3/CusdcV3Wrapper.sol
@@ -204,9 +204,9 @@ contract CusdcV3Wrapper is ICusdcV3Wrapper, WrappedERC20,
CometHelpers {

             rewardsAddr.claimTo(address(underlyingComet), address(this),
address(this), true);

-            uint256 bal = IERC20(rewardERC20).balanceOf(address(this));
+            uint256 bal = rewardERC20.balanceOf(address(this));
             if (owed > bal) owed = bal;
-            IERC20(rewardERC20).safeTransfer(dst, owed);
+            rewardERC20.safeTransfer(dst, owed);
         }
         emit RewardsClaimed(rewardERC20, owed);
```

```
    }
```

Additional recommendations

Reward token and underlying token must be different

The accounting logic in the *RewardableERC20* contract breaks down if its **rewardToken** is equal to the **underlying** in *RewardableERC20Wrapper* or equal to **_asset** in *RewardableERC4626Vault*. It is recommended to add checks to enforce the inequality.

Changes in *RewardableERC20Wrapper*:

```
    constructor(
        IERC20Metadata _underlying,
        string memory _name,
        string memory _symbol,
        IERC20 _rewardToken
    ) ERC20(_name, _symbol) RewardableERC20(_rewardToken, _underlying.decimals()) {
+         require(address(_rewardToken) != address(_underlying), "reward and underlying
are the same");
        underlying = _underlying;
        underlyingDecimals = _underlying.decimals();
    }
```

Changes in *RewardableERC4626Vault*:

```
    constructor(
        IERC20Metadata _asset,
        string memory _name,
        string memory _symbol,
        ERC20 _rewardToken
    )
        ERC4626(_asset, _name, _symbol)
        RewardableERC20(_rewardToken, _asset.decimals() + _decimalsOffset())
-     {}
+     {
+         require(address(_rewardToken) != address(_asset), "reward and asset are the
same");
+     }
```

Non-self-referential collaterals should add defaultThreshold check

The following comment says **defaultThreshold == 0** is used to create *SelfReferentialCollateral*:

```
uint192 defaultThreshold; // {1} A value like 0.05 that represents a deviation tolerance
// set defaultThreshold to zero to create SelfReferentialCollateral
```

Setting **defaultThreshold = 0** means that the that there is 0 tolerance in either direction, so if the market price just moves 1 wei away from **FIX_ONE**, a depeg is registered. For all non-self-referential collaterals, a **config.defaultThreshold > 0** check should be added (*CToken*, *LSDs*, etc).

Currently, only *CurveStableCollateral* has this check:

```
constructor(
    CollateralConfig memory config,
    uint192 revenueHiding,
    PTConfiguration memory ptConfig
) AppreciatingFiatCollateral(config, revenueHiding) PoolTokens(ptConfig) {
    require(config.defaultThreshold > 0, "defaultThreshold zero");
}
```

Raise version to 3.1.0

The audited version of the protocol is 3.1.0. Therefore, the **VERSION** variable in *Versioned* and the **ASSET_VERSION** variable in *VersionedAsset* should be set to this version.

Set exposedReferencePrice to underlyingRefPerTok in case of default

This recommendation applies to *AppreciatingFiatCollateral*, *CurveStableCollateral* and *CTokenV3Collateral*.

Currently, in the case that **underlyingRefPerTok < exposedReferencePrice**, **exposedReferencePrice** is set to **hiddenReferencePrice**. This is suboptimal since thereby the **exposedReferencePrice** will suddenly drop by the **revenueHiding** percentage in the case of a default.

```
uint192 underlyingRefPerTok = _underlyingRefPerTok();

// {ref/tok} = {ref/tok} * {1}
uint192 hiddenReferencePrice = underlyingRefPerTok.mul(revenueShowing);

// uint192(<) is equivalent to Fix.lt
if (underlyingRefPerTok < exposedReferencePrice) {
    exposedReferencePrice = hiddenReferencePrice;
    markStatus(CollateralStatus.DISABLED);
} else if (hiddenReferencePrice > exposedReferencePrice) {
    exposedReferencePrice = hiddenReferencePrice;
}
```

To have a smoother behavior of the **exposedReferencePrice** it is recommended to set it to **underlyingRefPerTok** instead.

Use break instead of continue in BasketHandler.quoteCustomRedemption() function

The *quoteCustomRedemption()* function uses the following loop to search for the index of **b.erc20s[j]** in **erc20sAll**:

```
uint256 erc20Index = type(uint256).max;
for (uint256 k = 0; k < len; ++k) {
    if (b.erc20s[j] == erc20sAll[k]) {
        erc20Index = k;
        continue;
    }
}
```

When the index is found, the **continue** statement is executed. This however won't save any Gas as the loop will just keep executing. To stop executing the loop, the **break** statement should be used instead.

Remove erc20 from Transfer struct in Distributor.distribute() function

erc20 is constant per function call. This means it does not need to be saved for each **Transfer** individually. It can be removed from the **Transfer** struct.

Stop execution of Distributor.distribute() function if tokensPerShare is zero

tokensPerShare can be equal to zero due to rounding:

```
RevenueTotals memory revTotals = totals();
uint256 totalShares = isRSR ? revTotals.rsrTotal : revTotals.rTokenTotal;
require(totalShares > 0, "nothing to distribute");
tokensPerShare = amount / totalShares;
```

This means that the *distribute()* function just continues executing zero value transfers. It is recommended to instead **return** or **revert** if **totalShares = 0** to save gas.

SDaiCollateral does not need revenue hiding

By inspection of the *Pot* contract it is clear that *pot.chi()*, i.e. the **refPerTok** rate that is used in the *underlyingRefPerTok()* function, can never decrease.

chi is initialized in the constructor and then only changed in the *pot.drip()* function where it cannot decrease:

```
function drip() external note returns (uint tmp) {
    require(now >= rho, "Pot/invalid-now");
    tmp = rmul(rpow(dsr, now - rho, ONE), chi);
    uint chi_ = sub(tmp, chi);
    chi = tmp;
    rho = now;
    vat.suck(address(vow), address(this), mul(Pie, chi_));
 }
```

Notice that **tmp** (the new **chi**) must be greater or equal to the old **chi**, otherwise the subtraction reverts.

The recommendation is to implement the *SDaiCollateral* like *StargatePoolFiatCollateral* by inheriting from *FiatCollateral* instead of *AppreciatingFiatCollateral*. This removes the overhead that is introduced by checking whether **refPerTok** has decreased.

Consider calling TradeLib.maxTradeSize() function with sellHigh price

The *TradeLib.maxTradeSize()* function is only ever called from within *TradeLib.prepareTradeSell()* with the **sellLow** price as parameter.

Using **sellLow** as the **price** parameter does not properly restrict the trade volume as the actual price most likely lies in between **sellLow** and **sellHigh**.

The **sellLow** price is used in this iteration of the protocol instead of the **sellHigh** price since the **sellHigh** price now decays toward **3*sellHigh** and then becomes **FIX_MAX**. The argument is that it's better to sell an increasing amount instead of a decreasing amount and then nothing.

For the *RevenueTrader* contract selling a decreasing amount during the decay phase could be better than selling an increasing amount. That's because this is the phase when there's uncertainty about the market price but still there's the hope of getting a market price in the future. When the decay phase ends without a new price being observed it makes sense to special case **sellHigh = FIX_MAX** and then to sell everything.

On the other hand, for the *BackingManager* it's more sensible to sell an increasing amount during the decay phase as an asset won't be sold at all when the decay phase has ended and **sellLow = 0**. Selling a decreasing amount could therefore lead to worse outcomes in the recollateralization algorithm by taking a bigger haircut.

The above shows there's no clear answer for which price to use. Our recommendation is to stick to using the **sellLow** price to ensure efficient recollateralization. In other words, the *BackingManager* is more important than the *RevenueTrader*.

The purpose of this recommendation is to lay out the tradeoff and to reason through it as it was specifically requested by the team.

BasketHandler.refreshBasket() can use stored lastStatus

It is not necessary to call *status()* since the call to *AssetRegistry.refresh()* already sets **lastStatus** to the current result of *status()*.

Applying the following change saves one iteration over all assets:

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

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

        trackStatus();
    }
```

BasketHandler.disableBasket() should call trackStatus()

The *BasketHandler.disableBasket()* function should call *BasketHandler.trackStatus()* to emit the **BasketStatusChanged** event in case the basket status has changed and to correctly set **lastStatus** and **lastStatusTimestamp**.

```
    function disableBasket() external {
        require(_msgSender() == address(assetRegistry), "asset registry only");

        uint256 len = basket.erc20s.length;
        uint192[] memory refAmts = new uint192[](len);
        for (uint256 i = 0; i < len; ++i) refAmts[i] = basket.refAmts[basket.erc20s[i]];
        emit BasketSet(nonce, basket.erc20s, refAmts, true);
        disabled = true;
+       trackStatus();

    }
```

Currently there is no impact since **lastStatus** and **lastStatusTimestamp** are never queried in this inconsistent state.

In WrappedERC20, implement decimals function to default to 18 decimals

This comment states that the *WrappedERC20* contract should default to 18 decimals if the child contract does not override it.

However currently the *decimals()* function is not implemented. Therefore, the recommendation is to implement this function to make the contract behave as stated in the comment.

In ICusdcV3Wrapper, Incorrect ordering of variables in UserBasic struct

The **UserBasic** struct that's defined in *ICusdcV3Wrapper* is wrong. It needs to have the **baseTrackingAccrued** and **baseTrackingIndex** variables swapped.

Currently the struct is not used, so there's no issue. However, it should be changed to avoid any issues in the future.

Remove unnecessary code in CTokenV3Collateral

The **CometCollateralConfig** struct should be removed as it is never used. Similarly the *bal()* function is not needed as it is implemented in exactly the same way in the *Asset* parent contract.

Use a more efficient function to claim rewards in CTokenWrapper

Currently the *CTokenWrapper* uses the *Comptroller.claimComp()* function with a single **holder** parameter to claim rewards. This is inefficient since this iterates over all Compound markets and claims rewards for supplying and borrowing.

It is recommended to call this function instead and pass the required parameters to only claim rewards for the specific **cToken** that is wrapped and only for supplying as the *CTokenWrapper* is only a supplier, not a borrower.

Deployment scripts have excessively long timeouts for Base

In *utils.ts*, *oracleTimeout()* returns an excessively long timeout when **chainId** is not equal to Ethereum's chain id. Since Reserve is going to be deployed on Base, this would lead to all collateral on Base to be deployed with **longOracleTimeout**.

```
export const longOracleTimeout = bn('4294967296')

// Returns the base plus 1 minute
export const oracleTimeout = (chainId: string, base: BigNumberish) => {
  return chainId == '1' ? bn('60').add(base) : longOracleTimeout
}
```

Thereby, collateral contracts on Base never time out even when their oracles are returning very stale prices.

Incorrect targetName for StargatePoolETHCollateral deployment

targetName should be ETH instead of USD for the *StargatePoolETHCollateral* deployment.

```
  const            collateral           =            <StargatePoolETHCollateral>await
StargateCollateralFactory.connect(
    deployer
  ).deploy({
    priceTimeout: priceTimeout.toString(),
    chainlinkFeed: networkConfig[chainId].chainlinkFeeds.ETH!,
    oracleError: fp('0.005').toString(), // 0.5%,
    erc20: erc20.address,
    maxTradeVolume: fp('1e6').toString(), // $1m,
    oracleTimeout: oracleTimeout(chainId, '3600').toString(), // 1 hr,
-   targetName: hre.ethers.utils.formatBytes32String('USD'),
+   targetName: hre.ethers.utils.formatBytes32String('ETH'),
    defaultThreshold: 0,
    delayUntilDefault: 0,
  })
```

Incorrect erc20 for maWETH deployment

The maWETH collateral plugin is deployed with **maWBTC.address** as the **erc20** address. It should be **maWETH.address** instead.

```
   const collateral = await SelfReferentialFactory.connect(deployer).deploy(
      {
        priceTimeout: priceTimeout,
        oracleError: fp('0.005'),
        maxTradeVolume: fp('1e6'), // $1m,
        oracleTimeout: oracleTimeout(chainId, '3600'), // 1 hr
        targetName: ethers.utils.formatBytes32String('ETH'),
        defaultThreshold: fp('0.05'), // 5%
        delayUntilDefault: bn('86400'), // 24h
        chainlinkFeed: networkConfig[chainId].chainlinkFeeds.ETH!,
-       erc20: maWBTC.address,
+       erc20: maWETH.address,
      },
      revenueHiding
    )
    assetCollDeployments.collateral.maWETH = collateral.address
    deployedCollateral.push(collateral.address.toString())
  }
```

maStETH plugin address is assigned incorrectly

The address of the **maWBTC** plugin is overwritten by the address of the **maStETH** plugin instead of making the correct assignment.

```
    const collateral = await NonFiatCollateralFactory.connect(deployer).deploy(
      {
        priceTimeout: priceTimeout,
        oracleError: combinedOracleErrors,
        maxTradeVolume: fp('1e6'), // $1m,
        oracleTimeout: oracleTimeout(chainId, '3600'), // 1 hr
        targetName: ethers.utils.formatBytes32String('ETH'),
        defaultThreshold: fp('0.01').add(combinedOracleErrors), // ~1.5%
        delayUntilDefault: bn('86400'), // 24h
        chainlinkFeed: networkConfig[chainId].chainlinkFeeds.ETH!,
        erc20: maStETH.address,
      },
      revenueHiding,
      networkConfig[chainId].chainlinkFeeds.stETHETH!, // {target/ref}
      oracleTimeout(chainId, '86400').toString() // 1 hr
    )
-   assetCollDeployments.collateral.maWBTC = collateral.address
+   assetCollDeployments.collateral.maStETH = collateral.address
    deployedCollateral.push(collateral.address.toString())
  }
```

Wrong oracle timeout configurations

The oracle timeouts that are configured for the asset plugins should be consistent with the Chainlink heartbeat times. This is recognized by Reserve as the deployment scripts mostly follow this rule.

However, there are exceptions as all *MorphoFiatCollateral* plugins are deployed with a timeout of 24 hours even though DAI has a heartbeat time of 1 hour. This means that it takes 23 hours longer than necessary to detect a timeout.

The recommendation is to check all oracle timeouts for instances of this issue. This must be done anyways to ensure support for Base as heartbeat times for Base and Ethereum can be different.

Note that based on issue *M-14*, the oracle timeout must not be set to exactly the heartbeat time, but instead should be slightly higher.

Enforce minimum votingDelay in Governance contract

If the **votingDelay** is zero or very small, users might not have sufficient time after a *StRSR* era change to stake **rsr** and secure themselves voting power. This means the **rToken** becomes prone to adversaries trying to take over a large portion of the voting power to then propose and execute malicious proposals.

A reasonable minimum could be 1 day. It can be checked in the constructor of the *Governance* contract.

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 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 or that have an increased risk of being hacked. 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.