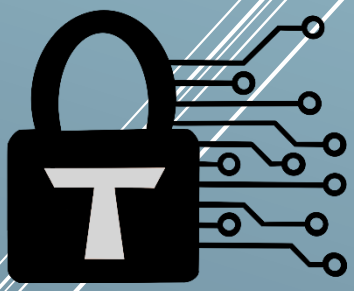
Trust Security

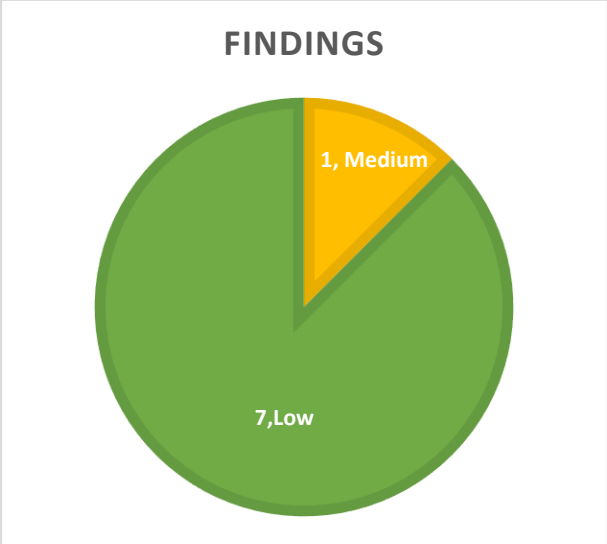


Smart Contract Audit

Reserve Protocol – 3.4.0 Upgrade Spell

24/05/24

Executive summary

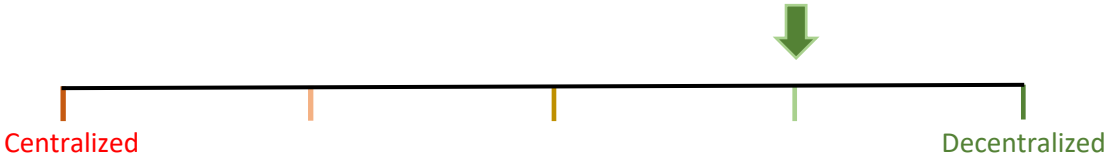


Category	Stablecoin
Auditor	HollaDieWaldfee gjaldon

Findings

Severity	Total	Fixed	Open	Acknowledged
High	-	-	-	-
Medium	1	1	-	-
Low	7	5	-	2

Centralization score



Signature

Digitally signed by Trust
Date: 2024.05.24 21:49:51 +03'00'

Document properties

Versioning

Version	Date	Description
0.1	24/05/2024	Client report

Contact

Trust

trust@trust-security.xyz

Introduction

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

Scope

The following file is in scope of the audit:

- 3_4_0.sol

In addition to the file above, the scope of the audit has been defined as follows:

- Check that it is possible for any RToken configuration to upgrade from version **3.0.0**/**3.0.1** to version **3.4.0**.
- The hardcoded asset addresses in the *3_4_0.*sol spell contract may not be compatible or have the intended consequences for all possible RToken deployments. Only the live deployments for **eUSD**, **ETH+**, **hyhUSD** (Ethereum), **USDC+**, **USD3**, **rgUSD**, **hyhUSD** (Base), **bsdETH**, **iUSDC**, **Vaya** and **MAAT** have been checked.
- The scope of the audit does not include upgrading from any version **<3.4.0** and **>=3.0.0** to **3.4.0**. This means the diffs from version **3.0.0**/**3.0.1** to version **3.4.0** have been checked but not the intermediate diffs (e.g., **3.1.0 -> 3.4.0**).

Repository details

- **Repository URL:** https://github.com/reserve-protocol/protocol
- **Commit hash:** b700311009300d5a662a1899f6cc83f4283248b5
- **Mitigation hash:** a353006f6e4e56b1bc4100aeae1e4c371c2af3a0

About Trust Security

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

About the Auditors

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

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.

Findings

Medium severity findings

TRST-M-1 Unregistering collateral that is not in the reference basket can cause loss of funds

- **Category:** Logical issues
- **Source:** 3_4_0.sol
- **Status:** Fixed

Description

castSpell2() unregisters all collateral plugins that are not in the reference basket and no longer supported in **3.4.0**.

```
        Registry memory reg = assetRegistry.getRegistry();
        for (uint256 i = 0; i < reg.erc20s.length; i++) {
            IERC20 erc20 = reg.erc20s[i];
            if (!reg.assets[i].isCollateral()) continue; // skip pure assets


            if (
                rotations[erc20] != IAsset(address(0)) ||
                (assets[erc20] == IAsset(address(0)) && basketHandler.quantity(erc20)
== 0)
            ) {
                // unregister rotated assets and non-3.4.0 assets not in the reference
basket
                assetRegistry.unregister(reg.assets[i]);
            }
        }
```

There are multiple possibilities for how unregistering collateral that is not in the reference basket can be unsafe, beyond the cases that are already documented in the code.

1) The collateral can be earned as a reward, and to unregister the collateral means to lose access to the rewards that have been earned already and might be earned in the future.
2) *castSpell2()* can be called when the RToken is undergoing recollateralization. In other words, the collateral by which it is backed is not the same as the collateral in the reference basket. As a result, the RToken can lose access to its backing, and take a haircut. The haircut is a loss to RToken holders, StRSR stakers or both.

Recommended mitigation

For (1), it is sufficient to add documentation as the scenario can be taken care of by correctly configuring the RToken instance prior to casting the spell.

For (2), it is recommended to check *basketHandler.fullyCollateralized()* in the *castSpell2()* function. This scenario is independent of RToken configurations and can occur at any moment based on on-chain conditions, e.g., when a collateral plugin in the reference basket depegs.

Team response

Added documentation here and here.

Added *fullyCollateralized()* check here.

Mitigation review

The recommendations have been implemented. Specifically, calling *castSpell2()* can no longer cause a haircut to be taken.

Low severity findings

TRST-L-1 Rotated collateral can be part of backup configs

- **Category:** Logical issues
- **Source:** 3_4_0.sol
- **Status:** Fixed

Description

Rotated erc20s are replaced in the prime basket.

```
            for (uint256 i = 0; i < primeERC20s.length; i++) {
                if (rotations[primeERC20s[i]] != IAsset(address(0))) {
                    primeERC20s[i] =
IERC20(address(rotations[primeERC20s[i]].erc20()));
                    newBasket = true;
                }
            }
```

However, the rotated erc20s can also be part of backup configs. This means they can still end up in the reference basket. *castSpell2()* would then try to unregister them.

```
            if (
                rotations[erc20] != IAsset(address(0)) || …)
            ) {
                // unregister rotated assets and non-3.4.0 assets not in the reference
basket
                assetRegistry.unregister(reg.assets[i]);
            }
```

The basket then becomes unsound, and the transaction reverts.

```
require(main.basketHandler().status() == CollateralStatus.SOUND, "basket not sound");
```

Recommended mitigation

Rotated erc20s should not only be replaced in the prime basket but also in the backup configs.

Team response

Fixed.

Mitigation review

The recommendation has been implemented. Note that *castSpell2()* can still unregister deprecated collateral plugins that are in the backup configs or prime basket but not in the reference basket. RToken deployments can thus end up in a state where they have no collateral plugins to fall back on in case a collateral in the reference basket becomes **DISABLED**. It is not the expectation that the spell must take care of all such possibilities. Instead, each RToken governance must simulate the effects of applying the spell and check which plugins will be deprecated.

TRST-L-2 Reweightable RTokens revert when updating the prime basket

- **Category:** Logical issues
- **Source:** 3_4_0.sol
- **Status:** Fixed

Description

The **reweightable** flag has been introduced in **3.2.0**. To make the spell as general as possible, it should support reweightable RTokens even though it is not needed for the current live deployments.

When swapping assets in the *castSpell1()* function, the basket can become disabled. And when the basket is disabled it won't be possible to normalize target amounts and *setPrimeBasket()* reverts.

Recommended mitigation

Call *forceSetPrimeBasket()* instead of *setPrimeBasket()*. The spell doesn't change target amounts, so bypassing target amount normalization for reweightable RTokens is safe.

Team response

Fixed.

Mitigation review

The recommendation has been implemented.

TRST-L-3 Proposals in old TimelockController can be executed without delay

- **Category:** Logical issues
- **Source:** 3_4_0.sol
- **Status:** Fixed

Description

The governance of each RToken deployment consists of the *Governance* contract and the *TimelockController*. The spell only upgrades to a new *Governance* contract, reusing the old *TimelockController*. After the upgrade, the two contracts will be out of sync in terms of the proposals that they are aware of. *TimelockController* has seen proposals that *Governance* has not seen.

Assuming *TimelockController* has seen a proposal and has executed it. *Governance* will then be able to create a new proposal with the same id but when it comes to scheduling / executing it, *TimelockController* will revert.

On the other hand, if *TimelockController* has seen a proposal and NOT executed it, *Governance* is then able to create a new proposal with the same id and execute it immediately without another delay.

```
    function execute(
        address[] memory targets,
```

```
        uint256[] memory values,
        bytes[] memory calldatas,
        bytes32 descriptionHash
    ) public payable virtual override returns (uint256) {
        uint256 proposalId = hashProposal(targets, values, calldatas,
descriptionHash);


        ProposalState currentState = state(proposalId);
        require(
>           currentState == ProposalState.Succeeded || currentState ==
ProposalState.Queued,
            "Governor: proposal not successful"
        );
        _proposals[proposalId].executed = true;


        emit ProposalExecuted(proposalId);


        _beforeExecute(proposalId, targets, values, calldatas, descriptionHash);
        _execute(proposalId, targets, values, calldatas, descriptionHash);
        _afterExecute(proposalId, targets, values, calldatas, descriptionHash);


        return proposalId;
    }
```

Once the votation period has passed and was successful, the proposal can be executed immediately since its state will be **Succeeded**.

It is possible that the old proposal was made in a different **era**, and proposals in one **era** should not have an impact on later **eras**.

The scenarios described above have been confirmed with unit tests.

Recommended mitigation

In addition to upgrading to a new *Governance* contract, the spell should also use a new *TimelockController* contract to start with a fresh governance setup.

Team response

Initially fixed here, further mitigations applied here and here.

Mitigation review

The recommendation has been implemented. All new *Governance* and *TimelockController* contracts have been checked and are set up correctly.

TRST-L-4 Spell contains collateral plugins that are still in development

- **Category:** Configuration issues
- **Source:** 3_4_0.sol
- **Status:** Fixed

Description

The spell contains addresses for the **cvxeUSDFRAXBP**, **cvxETHPlusETH** and **crveUSDFRAXBP** collateral plugins which are still undergoing an audit and can't be used for live deployments yet.

Recommended mitigation

Remove the affected plugin addresses or add comments that RTokens using **cvxeUSDFRAXBP**, **cvxETHPlusETH** or **crveUSDFRAXBP** can't use the spell yet.

Team response

Fixed here and here.

Mitigation review

TODOs have been added to the affected plugin addresses. RToken deployments for which the affected plugins are relevant need to wait until the TODOs are resolved.

TRST-L-5 New rewards that are earned for deprecated collaterals can be lost

- **Category:** Logical issues
- **Source:** 3_4_0.sol
- **Status:** Fixed

Description

For some collateral plugins only **msg.sender** is able to claim rewards. But *RewardableLib* now performs a **delegatecall** into the collateral plugin which, if it is older than **3.3.0**, might not have a *claimRewards()* function implemented.

The same problem with rewards occurs if the collateral plugin is not unregistered but the reward asset (must be a **collateral** as well, since only **collaterals** are unregistered in *castSpell2()*) is unregistered.

The expectation should be that after calling *castSpell2()*, all new rewards that are accrued for collateral plugins prior to **3.4.0** can be lost.

Recommended mitigation

The behavior should be documented such that RToken governances are aware of it. Trying to support both reward claiming with **delegatecall** and without **delegatecall** and to keep track of reward tokens on the protocol level is over-engineering.

Team response

Fixed.

Mitigation review

The recommendation has been implemented and the behavior is documented.

TRST-L-6 The initBal check in GnosisTrade introduces griefing concern for tokens with high decimals

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

Description

The **initBal <= type(uint96).max** check in *GnosisTrade* is not necessary since **gnosis** will at most use **req.sellAmount** of tokens which is already checked to be **<= type(uint96).max**.

In the worst case, the redundant check can introduce a griefing concern where an attacker can pre-fund the contract such as to make the check fail. **2^96 = ~8e28**, so it would require **8e10** tokens for an ERC20 with 18 decimals in the *GnosisTrade* contract for the check to fail. Even considering that a substantial amount of the funds can be provided by the protocol, it is highly unlikely that an attacker would attempt this and almost impossible with the tokens that Reserve currently supports (mostly tokens pegged to USD and ETH and their derivatives).

A minor note is that the approval can be made for **req.sellAmount** instead of **initBal** since the **gnosis** contract only needs to transfer **req.sellAmount**.

Recommended mitigation

Update the approval amount and remove the **initBal <= type(uint96).max** check.

```
--- a/contracts/plugins/trading/GnosisTrade.sol
+++ b/contracts/plugins/trading/GnosisTrade.sol
@@ -94,7 +94,6 @@ contract GnosisTrade is ITrade, Versioned {
         initBal = sell.balanceOf(address(this)); // {qSellTok}
         sellAmount = shiftl_toFix(initBal, -int8(sell.decimals())); // {sellTok}

-        require(initBal <= type(uint96).max, "initBal too large");
         require(initBal >= req.sellAmount, "unfunded trade");

         assert(origin_ != address(0));
@@ -139,7 +138,7 @@ contract GnosisTrade is ITrade, Versioned {
         //
         // Context: wcUSDCv3 has a non-standard approve() function that reverts if
the approve
         // amount is > 0 and < type(uint256).max.
-        AllowanceLib.safeApproveFallbackToMax(address(sell), address(gnosis),
initBal);
+        AllowanceLib.safeApproveFallbackToMax(address(sell), address(gnosis),
req.sellAmount);
```

Team response

The finding will be addressed in version **4.0.0**. More broadly, in version **4.0.0**, the compatibility of *GnosisTrade* with tokens that have more than 18 decimals must be assessed.

TRST-L-7 Updated reward ratios for Furnace and StRSR are only approximations

- **Category:** Math issues
- **Source:** 3_4_0.sol

- **Status:** Acknowledged

Description

In version **3.0.0**/**3.0.1**, rewards ratios in *StRSR* and *Furnace* are set based on a payout per block. In version **3.4.0**, rewards are paid out per second, and the old reward ratios need to be adjusted. To convert the per-block ratio to a per-second ratio, the per-block ratio is divided by 12 on Ethereum and divided by 2 on Base.

```
        {
            uint48 blocktime = mainnet ? 12 : 2;
            proxy.furnace.setRatio(proxy.furnace.ratio() / blocktime);
            TestIStRSR(address(proxy.stRSR)).setRewardRatio(
                TestIStRSR(address(proxy.stRSR)).rewardRatio() / blocktime
            );
        }
```

Reward is not paid out linearly, which means that dividing the old reward ratio by 12 or 2 is only an approximation.

Let **x** be the reward ratio before the upgrade, e.g., **0.05 = 5% per block**.

Let **y** be the reward ratio after the upgrade, e.g., **0.05 = 5% per second**.

To get exact solutions, the following must be solved for **y** for deployments on Base:

1 – x = (1 – y)^2.

And for deployments on Ethereum, the following must be solved for **y**:

1 – x = (1 – y)^12.

The reward ratio that is calculated as **y = x/2** and **y = x/12** is a bit smaller than intended.

The deviations can be visualized in a graph. **x** is the number of seconds for which to accrue rewards. **g(x)** gives the percentage by which rewards paid out are underestimated for a deployment on Base. **h(x)** is the same for Ethereum. Both functions are parameterized by the per-block rate **r1**.

Recommended mitigation

The deviation is negligible for any reasonable reward ratios **r1** and can be acknowledged.

Team response

The finding has been acknowledged.

Mitigation review

Since the deviation is expected to be small, it is fair to acknowledge the finding. Any RToken instance for which the deviation is a concern can update the reward ratios after *castSpell1()* has been called by proposing such an update in the new *Governance* contract.

Additional recommendations

Wrong error message when Alexios does not have PROPOSER_ROLE

The check that the **Alexios** *Governance* contract has the **PROPOSER_ROLE** reverts with a wrong error message.

In the mitigation commit, the check is no longer needed. All error messages are now correct.

Document invariant that assets[erc20] != address(0) must imply rotations[erc20] = address(0)

The spell logic assumes that an **erc20**, for which **assets[erc20] != address(0)**, does not have a rotation configured, i.e., **rotations[erc20] = address(0)**.

While the invariant does currently hold, it can be easily enforced in the contract to avoid misconfigurations in the future.

Verify WETH collateral plugin on Base

The WETH collateral plugin on Base is missing verification on Basescan.

Unregistering plugins before setting prime basket or backup config can cause revert

TUSD is unregistered before the prime basket (and with mitigations applied the backup configs) is set. If in either of them there is an erc20 which does not have a collateral plugin registered, the transaction reverts.

This is not a problem for **eUSD**, which is the only RToken deployment that has **TUSD** in its backup config. For **eUSD**, the backup config is not changed, and so the check that causes the revert is never made.

Still, the pattern of unregistering collateral plugins before setting the prime basket and backup configs should be avoided. Doing it the other way around, setting prime basket and backup configs first, is the correct pattern to adopt.

STG and USDbC should be deprecated but they have a plugin for 3.4.0

Based on communication with the client, the STG asset and USDbC collateral should be deprecated. However, the spell has a hardcoded address for both plugins. Since both plugins should be deprecated, their addresses should be removed from the spell.

Rotation for fUSDC uses collateral address instead of erc20 address

The rotation for **fUSDC** uses the collateral address instead of the erc20 address. There should not be a rotation at all for the plugin at **0x3C0a9143063Fc306F7D3cBB923ff4879d70Cf1EA** since it already uses the plain **fUSDC** erc20. So, it should just update the plugin without updating the erc20, which is already correctly configured. There is no impact to this finding and creating an entry for a collateral plugin address in the **rotations** mapping is a no-op.

Order of swapping RTokenAsset does not matter

In a comment, it is stated that the new *RTokenAsset* should be registered as the last plugin. Following the code that is executed by calling *AssetRegistry.swapRegistered(rTokenAsset)*, none of it depends on the other assets registered in the *AssetRegistry*. So, the comment stating that *RTokenAsset* might depend on other assets in the *AssetRegistry* is not needed.

Broker.priceNotDecayed() function does not behave as expected by its name

The *Broker.priceNotDecayed()* function has the effect that a *DutchTrade* can only be created for a buy / sell asset if the price of the asset has been refreshed in the same block.

```
    /// @return true iff the price is not decayed, or it's the RTokenAsset
    function priceNotDecayed(IAsset asset) private view returns (bool) {
        return asset.lastSave() == block.timestamp || address(asset.erc20()) ==
address(rToken);
    }
```

The intention is to only start Dutch trades if the pricing is reliable such that the protocol doesn't incur a loss from wrong pricing of the assets.

But the name of the function ("not decayed") suggests a different behavior, which is either of the two:

1. price decay has not started yet (price is up to date or in the decay delay)
2. price decay is not finished, i.e., price has not decayed to **0 / FIX_MAX** (price is up to date, in the decay delay, or in the price timeout)

It has been determined that in release **4.0.0**, the name of the function should be changed.

Only Broker needs to call cacheComponents()

Since all RToken deployments that are supposed to be upgraded have at least version **3.0.0**, caching components for contracts other than *Broker* is redundant because the components have already been cached in an upgrade from **<3.0.0** to **3.0.0**/**3.0.1**.

Reuse existing variables

Some of the components are saved to two different variables. Reuse existing variables instead.

```
@@ -287,11 +287,6 @@ contract Upgrade3_4_0 {
                 Components memory compImpls,
                 TradePlugins memory tradingImpls
             ) = deployer.implementations();
-            IBackingManager backingManager = main.backingManager();
-            IBroker broker = main.broker();
-            IDistributor distributor = main.distributor();
-            IRevenueTrader rTokenTrader = main.rTokenTrader();
-            IRevenueTrader rsrTrader = main.rsrTrader();

             UUPSUpgradeable(address(main)).upgradeTo(address(mainImpl));
             UUPSUpgradeable(address(proxy.assetRegistry)).upgradeTo(
@@ -312,15 +307,15 @@ contract Upgrade3_4_0 {

UUPSUpgradeable(address(proxy.rToken)).upgradeTo(address(compImpls.rToken));

             // Trading plugins
-
TestIBroker(address(broker)).setDutchTradeImplementation(tradingImpls.dutchTrade);
-
TestIBroker(address(broker)).setBatchTradeImplementation(tradingImpls.gnosisTrade);
+
TestIBroker(address(proxy.broker)).setDutchTradeImplementation(tradingImpls.dutchTrade
);
+
TestIBroker(address(proxy.broker)).setBatchTradeImplementation(tradingImpls.gnosisTrad
e);

             // cacheComponents()
-            ICachedComponent(address(broker)).cacheComponents();
-            ICachedComponent(address(backingManager)).cacheComponents();
-            ICachedComponent(address(distributor)).cacheComponents();
-            ICachedComponent(address(rTokenTrader)).cacheComponents();
-            ICachedComponent(address(rsrTrader)).cacheComponents();
+            ICachedComponent(address(proxy.broker)).cacheComponents();
+            ICachedComponent(address(proxy.backingManager)).cacheComponents();
+            ICachedComponent(address(proxy.distributor)).cacheComponents();
+            ICachedComponent(address(proxy.rTokenTrader)).cacheComponents();
+            ICachedComponent(address(proxy.rsrTrader)).cacheComponents();
         }
```

Use the **basketHandler** variable instead of querying the address again from **main**.

```
-                    require(main.basketHandler().status() == CollateralStatus.SOUND,
"basket not sound");
+                    require(basketHandler.status() == CollateralStatus.SOUND, "basket
not sound");
```

castSpell2() function does not need anastasius argument

Providing **anastasius** as an argument to *castSpell2()* is not needed and a leftover from a previous version of the contract.

Centralization risks

The *3_4_0.sol* spell contract does not introduce any new centralization risks. It is fully permissionless. The hardcoded addresses are set by Reserve. Once they have been confirmed by the RToken governance, and the upgrade has been simulated, Reserve cannot interfere with the upgrade in any way.

For the core protocol contracts and the plugins, the same risks apply that have been discussed in previous reports.

Systemic risks

Casting the spell can fail and its successful execution must not be relied on

Calling both the *castSpell1()* and *castSpell2()* functions can fail. There exist different conditions under which they can fail, for example:

- One of the new plugins that is registered is not **SOUND**.
- An **erc20** in either the prime basket or in a backup config has been unregistered prior to the upgrade such that *forceSetPrimeBasket()* or *setBackupConfig()* fails.

Importantly, a scenario like in the first example cannot be ruled out. This means that there must not be any downstream impact in the failure. For example, there might be two proposals made after one another such that the spell is expected to finish before the second proposal. The guardian may not track this second proposal since after *castSpell1()*, the *Governance* contract has changed, and the proposal would fail naturally. However, when casting the spell fails, the second proposal might be executed in an unintended and vulnerable state.

Spell assumes specific RTokens and RToken configurations

3_4_0.sol only supports a hardcoded set of RToken instances. Moreover, the supported RTokens are assumed to have a specific configuration. If the supported RTokens are changed from their current configuration, the spell can have unintended consequences. For example, plugins may not be rotated if new plugins are introduced before the upgrade. In terms of upgrading the core protocol contracts, the spell assumes to upgrade the contracts from version **3.0.0** or **3.0.1** to **3.4.0**. Upgrading from any version other than **3.0.0** or **3.0.1** may cause disruptions or even loss of funds. The upgrade should be simulated before it is performed to ensure the RToken can continue its operation.

Unfinished proposals in old Governance contract cannot be executed after the Governance contract change

As part of *castSpell1()*, the RToken instance is transitioned to a new *Governance* and *TimelockController* contract. The **MAIN_OWNER_ROLE** is revoked from the old *TimelockController* and it is granted to the new *TimelockController*. As a result of this transition, any unfinished proposals in the old *Governance* contract that rely on **MAIN_OWNER_ROLE** to execute successfully, cannot be executed.