PUBLIC

# Code Assessment

## of the Smart Allocator Smart Contracts

June 10, 2025

Produced for

by

CHAINSECURITY

# Contents

# 1 Executive Summary

Dear all,

Thank you for trusting us to help Decentralized USD with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Smart Allocator according to Scope to support you in forming an opinion on their security risks.

Decentralized USD implements smart contracts for onboarding Real-World Assets as collateral for the USDD 2 system. The new contracts allow trusted RWA lenders to borrow USDD stablecoins against these assets, while managing debt repayment and liquidation.

The most critical subjects covered in our audit are access control, functional correctness, and precision of arithmetic operations. The general subjects covered are documentation, specifications, and gas efficiency.

Security regarding all aforementioned subjects is high.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

## 1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 0 |
| Low-Severity Findings | 3 |
| • Code Corrected | 1 |
| • Risk Accepted | 2 |

# 2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

## 2.1 Scope

The assessment was performed on the source code files inside the Smart Allocator repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

| V | Date | Commit Hash | Note |
|---|---|---|---|
| 1 | 19 May 2025 | 4cdb13ab06b4c2f91afe084d3367542aa4eebcd3 | Initial Version |
| 2 | 5 Jun 2025 | e2ee5a56e2926015ee007c53ece9d49110c568c7 | After Intermediate Report |

For the solidity smart contracts, the compiler version `0.6.12` was chosen. Note that the Tron solidity compiler is used. We assume that it behaves equivalently to the solidity 0.6.12 compiler while handling Tron-specific properties (e.g. precompile, address handling). Further, the compiler has never been subject of security reviews and, thus, could expose the system to a higher risk compared to the regular solidity compiler.

The following files were in scope:

```
src/
    auth/auth.sol
    conduits/
        RwaInputConduit2.sol
        RwaOutputConduit2.sol
        RwaSwapInputConduit.sol
        RwaSwapOutputConduit.sol
    gemjoins/join-auth.sol
    jars/RwaJar.sol
    oracles/RwaLiquidationOracle.sol
    spells/
        RwaLiquidationSpell_mainnet.sol
        RwaSpell_mainnet.sol
    tokens/
        RwaToken.sol
        RwaTokenFactory.sol
    urns/
        RwaUrn2.sol
        RwaUrnCloseHelper.sol
    value/value.sol
```

### 2.1.1 Excluded from scope

All other files are out of scope. The system's core contracts and other contracts, such as the PSM, are out of scope and assumed to be correct. Governance is expected to configure the contracts accordingly. Incorrect configuration may lead to issues.

Note that the hardcoded constants were validated against the USDD documentation (archive). For the undocumented contracts Vat and USDD, the values set in the UsddJoin have been read from the contract with Tronscan. Similarly, the USDT address has been read from the PSM's gem join's gem address. The correctness of the values documented and returned by Tronscan are out of scope.

## 2.2 System Overview

This system overview describes the latest received version of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Decentralized USD offers smart contracts for onboarding RWAs as collateral for USDD V2.

### 2.2.1 RWA Contracts

The Real-World Asset (RWA) contracts provide a dedicated infrastructure enabling the use of tokenized RWAs as collateral within the ecosystem. The system facilitates the issuance of USDD against these assets by designated trusted parties and incorporates specific mechanisms for debt management, stability fee handling, and liquidation.

**Locking Collateral and Drawing USDD.** The gem of RWA ilks corresponds to the `RwaToken` contract which is an ERC-20 with a fixed supply, deployable via the `RwaTokenFactory`. Since the supply is fixed, the oracle price should determine the value of the RWA reserves.

Once the token contract is created and tokens are transferred to a trusted destination (e.g., an operator), these funds can be used within the `RwaUrn2` contract (whitelisted for authenticated gem joins, as detailed below) to lock RWA collateral. This process enables the minting of USDD to an output conduit.

Specifically, operators have access to the following functionalities for managing the position:

- `lock`: Pulls an amount of gem tokens, joins them via the authenticated join adapter and converts the gem to ink.
- `free`: Converts the ink back to gem and exits gem tokens through the join adapter to the operator.
- `draw`: First, the interest rate is updated with `Jug.drip` to prevent unnecessary interest payment. Then, debt is generated, and USDD is minted through the USDD join adapter. The recipient of these funds is always the output conduit.
- `quit`: This function transfers the entire USDD balance to the output conduit. Notably, quit can be called by any address if the Vat is not live, indicating that the shutdown has been initiated.

The `wipe` function can be used permissionlessly to repay debt. It first updates the interest, then burns USDD via the UsddJoin adapter to settle the debt. Note that the function is permissionless as debt could be repaid by anyone through the Vat.

Further, `RwaUrn2` integrates with the `AuthGemJoin` to generate gem balances in the Vat. The authenticated gem join adapter provides the function `join`, which creates gem balances within the Vat while pulling funds from the callers (expected to only be the respective RWA urn). The `exit` function allows arbitrary addresses to exit gem balances as RWA tokens. Notably, the function is not privileged, ensuring that during system shutdown exits can be performed by arbitrary addresses.

Note that, despite representing a RWA, the RWA token does not formally provide a claim to the RWA assets.

Both the RWA urn and the join adapter provide the authenticated `rely` and `deny` functions to assign and revoke the governance role to and from given addresses. Similarly, the RWA urn provides governance with `hope` and `nope` to manage the operator role. Additionally, the RWA urn allows governance to update the following parameters with `RwaUrn2.file`:

- output conduit: destination of USDD when drawing/quitting
- jug: contract managing rate updates which should always correspond to the vat's jug.

The `RwaUrnCloseHelper` can be set as an operator on the urn to add functionality allowing to close an urn by wiping all debt. Namely, the `close` function estimates the debt to repay so that all collateral can be freed and calls `wipe` in the urn accordingly.

**Output Conduits.** Output conduits are the designated recipients for USDD generated by the RWA urns. Their core functions are designed for managing fund transfers.

The two main functions of the output conduit are:

- `pick`: Operators pick any bud (whitelisted by governance with `kiss` and `diss`) as the destination for the funds (not necessarily USDD is sent).
- `push`: "Mates" execute the fund transfers to the chosen destination address. After a transfer, the destination address is automatically reset to `0x0`, preventing further pushes until a new destination is set.

Note that the provided output conduits differ in their exact operation. Below is more detailed information on the implemented output conduits:

- `RwaOutputConduit2`: `push` pushes the full USDD balance held by the conduit to the destination.
- `RwaSwapOutputConduit`: The contract integrates with a PSM on `push` and, hence, converts the held USDD to the PSM's gem by swapping on the PSM. Additionally, `quit` is provided to mates and allows moving the held USDD to a `quitTo` address. Governance can choose the PSM and `quitTo` with `file`, and can withdraw any token from the conduit with `yank`.

**Input Conduits.** Input conduits are used to deposit tokens to repay outstanding debt. They primarily feature a `push` function, accessible by mates, which transfers the conduit's local token balance to a specified `to` address.

- `RwaInputConduit2`: Designed to receive USDD tokens. Its `push` function transfers the USDD balance to a pre-defined, immutable `to` address.
- `RwaSwapInputConduit`: Expects to receive the gem tokens of the configured PSM. When `push` is called, it sells these gem tokens for USDD and directs the resulting USDD to the `to` address. Mates can also use `quit` to retrieve gem tokens from the conduit and send them to a `quitTo` address. Governance can configure the PSM, `to`, and `quitTo` addresses using `file`. Furthermore, governance can assign/revoke governance and mate roles with `rely` / `deny` and `mate` / `hate` respectively, and can `yank` any token from the conduit.

Valid destinations for the input conduits include the RWA urn (for debt repayments) and the `RwaJar` which facilitates direct stability fee payments into the system's surplus. The RWA jar burns the received USDD to contribute Vat-internal usdd balances to the Vow, thereby building up a surplus buffer. It achieves this with the functions `void` and `toss`. The former uses the balance held by the jar to build up the surplus buffer while the latter pulls USDD from the caller.

**Liquidations and Price Oracle.** To price RWAs the system uses `DSValue`, allowing privileged addresses to update the prices with `poke`. However, unlike typical on-chain liquidations triggered by price changes, RWA liquidations are initiated manually by governance through the `RwaLiquidationOracle`. Specifically, the liquidation oracle provides governance with the functionality:

- `init`: Initializes an ilk on the liquidation oracle by deploying a `DSValue` contract (owned by the liquidation oracle) and poking it with the agreed-upon value, configuring the remediation period `tau` and publishing the hash of the off-chain agreement. For re-initialization, the remediation period and document hash can be updated. Note however, that the remediation period can only be lengthened, and not shortened.
- `bump`: Allows updating the underlying oracle, but strictly limits updates to increasing the RWA's value.

- `tell`: Initiates the liquidation process for an ilk, starting the remediation period during which debt should be repaid. Governances spells triggering `tell` should set ilk's line to zero, preventing further borrowing.
- `cure`: If the issue causing the liquidation has been resolved, governance can stop the liquidation. Governance spells stopping the liquidation should also consider restoring the line if necessary.
- `cull`: Once the remediation period ends and governance determines no further debt will be repaid, a write-off can be triggered which sets the RWA's value to zero and reclassifies any unpaid debt as system debt.
- `file`: Allows setting the Vow.

### 2.2.2 RWA Spells

Two RWA spell templates are provided as templates for governance spells.

**RWA Initialization Spell.** The `RwaSpell` serves as a template to initialize an RWA ilk.

Note that the common process of deployment and initialization is that an EOA deploys a contract and hands over the ownership of the deployment to the pause proxy. Then, the spell is deployed and the contracts and spell are carefully examined by governance before scheduling the spell execution. Eventually, the spell will be executed by the pause proxy through a delegatecall.

The `RwaSpell` performs the following actions:

1. Sanity checks on the constructor parameters of the relevant contracts.
2. Initialization of the RWA ilk on the Vat and Jug as well a debt ceiling configurations.
3. Configuration of DSValue as the oracle on the spot as well as poking the spot so that the value is published to the Vat.
4. Authorization of the authenticated gem join and liquidation oracle on the vat so that `slip` and `grab` can be called. Similarly, the RWA urn is authorized for the gem join.
5. One operator is assigned the operator and mate roles on the urn and the conduits. The `quitTo` is set as the urn for the output conduit and as the user for input conduits. Further, the user is set as the whitelisted recipient (bud) on the output conduit. Note that spell is expected to use swap conduits and non-swap conduits cannot be configured with it.

**Liquidation Spell.** The `RwaLiquidationSpell` implements a template for the governance-initiated liquidation process. Namely, it reduces the debt ceiling by the configured one. Then, the liquidation is initiated, and the debt is written off. Finally, the spot is notified about the new price (zero) and publishes the new price to the Vat.

### 2.2.3 Changelog

In Version 2, inconsistencies between `RwaOutputConduit2` and `RwaSwapOutputConduit` were resolved. In prior versions, the fund recipients could be arbitrary addresses in the `RwaOutputConduit2`, and `RwaSwapOutputConduit` could make all addresses mates and operators by assigning `0x0` the respective role.

## 2.3 Trust Model

The system implements several roles:

- Governance: Fully trusted. Governance could file malicious output conduits on urns or setup malicious operators that could drain the system (or configure a malicious PSM, yank the gem balances, ...). Further, governance is expected to validate and configure the deployment meaningfully. Note that governance is fully trusted in the core as it could maliciously mint stablecoins against bad ilks.

- Urn Operator: Trusted to respect the agreement by locking collateral and repaying debt eventually.
- Conduit Mates: Trusted to operate accordingly and push and quit whenever deemed necessary.
- Conduit Operators: Trusted to responsibly choose among whitelisted recipient addresses.
- Conduit Buds: Expected to be trusted addresses controlled by the RWA partner that respect the off-chain agreement.
- RWA partner: Fully trusted. The RWA partner has no incentive to repay the debt but is expected to respect off-chain agreements and repay debt according to the agreement. The reserves are expected to be monitored and governance is expected to take swift action in case of misbehavior. The damage is limited to the line that was configured.

Note that the full RWA system highly relies on trust and respect for off-chain agreements. Parameters should be carefully chosen to ensure that damage is limited so that the peg of USDD can be protected.

### 2.3.1 Changelog

In (Version 2), the trust model was further unified. In prior versions, Conduit Operators were trusted to pick only trusted addresses as destinations if no buds mapping was available.

# 3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

# 4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

| Likelihood | Impact | | |
|---|---|---|---|
| | High | Medium | Low |
| High | Critical | High | Medium |
| Medium | High | Medium | Low |
| Low | Medium | Low | Low |

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

# 5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 0 |
| Low-Severity Findings | 2 |



## 5.1 Governance Can Bypass Liquidation Delay Mechanism

Correctness Low Version 1 Risk Accepted

*CS-USDD-RWA-002*

In the RwaLiquidationOracle contract, the `tell` function initiates a repayment window (defined by `tau`) during which the borrower can repay their debt. If the repayment does not occur within this period, governance can call `cull` to write off the borrower's debt.

However, note that the repayment window check in `cull` ignores the case when `toc` is `0`:

```
require(block.timestamp >= add(ilks[ilk].toc, ilks[ilk].tau), "RwaOracle/early-cull");
```

Note that `toc` remains set to `0` while `tell` has not been called. Therefore, governance can bypass the remediation window by calling `cull` before `tell` due to `toc` being `0`.

Moreover, even if `tell` was called, the `cure` function can be used to reset `toc` to `0`. This also allows `cull` to bypass the grace period, effectively enabling governance to liquidate the position at any time.

---

**Risk accepted:**

Decentralized USD accepts the risk and states:

```
This behavior is controlled exclusively through the governance process. In practice, the expected flow is:

- tell → cure (repayment during the remediation period), or
- tell → cull (liquidation after the delay expires).

While it is technically possible to call cull before tell, this is not a concern under the assumed
governance model, where such actions are subject to transparent decision-making and community oversight.
Therefore, we consider this risk acceptable in the context of our governance-controlled operation.
```

## 5.2 Liquidation Spell Will Fail

Correctness | Low | Version 1 | Risk Accepted

*CS-USDD-RWA-003*

In the RwaLiquidationSpell contract, the `setup` function calls the `tell` function of RwaLiquidationOracle, initiating the liquidation process. Immediately afterward, it calls the `cull` function to liquidate the vault. However, `cull` can only be called after the time window defined by the ilk's `tau` variable, during which the debt can still be repaid, has elapsed. As a result, if this grace period is set to a nonzero value, the spell will revert.

---

**Risk accepted:**

Decentralized USD accepts the risk and states:

```
We have provided a liquidation spell template assuming tau = 0, where tell and cull are executed
within the same transaction. For cases where tau > 0, the tell and cull steps are separated and executed
in different spell flows to respect the required delay.
```

# 6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 0 |
| Low-Severity Findings | 1 |

- Frontrunning in Conduits Code Corrected

| | |
|---|---|
| Informational Findings | 1 |

- Code Inconsistencies Code Corrected

## 6.1 Frontrunning in Conduits

Security Low Version 1 Code Corrected

*CS-USDD-RWA-001*

There are griefing vectors in RwaSwapOutputConduit.

The `pick` function can effectively be called permissionlessly by setting `can[address(0)] == 1` via `hope(address(0))`. Since `pick` allows `to` to be set to `address(0)`, and `push` fails when `to == address(0)`, this enables frontrunning attacks that cause `push` calls to fail.

Additionally, the `onlyMate`-restricted functions `push` and `quit` can also be called permissionlessly by setting `may[address(0)] == 1` via `mate(address(0))`. An attacker can frontrun legitimate `push` calls by calling `push(uint256)` with a small amount, setting `to` to `address(0)` during execution and causing the subsequent call to revert.

These attacks are made cheaper and more feasible due to low transaction costs on the TRON chain.

---

**Code corrected:**

The `pick` function in RwaSwapOutputConduit was updated to eliminate the fallback behavior where any address could act as the operator if `address(0)` was set. This prevents front-runners from setting `to = address(0)` via `pick` to disrupt subsequent `push` calls.

Similarly, the `onlyMate` modifier was modified to prevent any address from acting as a mate when `address(0)` was set. This blocks frontrunning attackers from setting `to = address(0)` by calling `push` with small amounts. As a result, front-running attempts to trigger `push` reverts are no longer possible.

## 6.2 Code Inconsistencies

Informational Version 1 Code Corrected

*CS-USDD-RWA-004*

The codebase is inconsistent on several occasions. Below is a non-exhaustive list of inconsistencies:

- No-swap and a swap output conduits are provided. However, they unnecessarily differ in terms of access control:
    - The swap conduit whitelists all addresses as mates if `0x0` is a mate. Consequently, push and quit access can be granted to everyone for swap conduits. In contrast, the no-swap conduit does not support such functionality.
    - For the output conduits, the swap conduit requires the destination address `to`, picked by the operators, to be a whitelisted bud while the no-swap conduit does not require this.
- In most of the contracts the math functions revert with an error message. However, the math functions in `RwaToken` do not revert with an error message.

---

**Code corrected:**

1. The `onlyMate` modifier was updated to remove the logic that previously allowed any address to qualify if `address(0)` was set as a mate.
2. The `bud` mapping was reintroduced to manage access control. The `kiss` and `diss` functions were added to grant and revoke bud status, respectively. The `pick` function now includes a check to ensure that `who` is either a bud or equal to `address(0)`.
3. Not corrected. However, given that most of the inconsistencies have been resolved, we marked the issue as resolved as the remaining one has low relevance.

# 7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

## 7.1 Events Not Emitted

**Informational** **Version 1** **Acknowledged**

*CS-USDD-RWA-005*

Some contracts do not emit events consistently. The following is a non-exhaustive list of not emitted events:

- In `RwaToken`, the constructor should emit the `Transfer` event as per ERC20 standard. While not strictly required by the standard, emitting the event could be helpful for frontends and indexers.
- In `RwaSwapInputConduit`, the constructor should emit the `File` event since the PSM and the `to` address are initialized.
- In `RwaSwapOutputConduit`, the constructor should emit the `File` event since the PSM address is initialized.

---

**Acknowledged:**

Decentralized USD acknowledges the issue.

## 7.2 Gas Optimizations

**Informational** **Version 1** **Acknowledged**

*CS-USDD-RWA-006*

Below is non-exhaustive list of code sections that could be optimized to consume less gas:

- `RwaToken`: `totalSupply` could be declared as immutable since no further minting can occur.
- `RwaInputConduit2`: `usdd` and `to` could be declared as immutables.
- `RwaOutputConduit2`: `usdd` could be declared as immutable.
- `RwaLiquidationOracle`: `vat` could be declared as immutable.
- `RwaUrn2`: `vat`, `gemJoin` and `usddJoin` could be declared as immutables. Similarly, `gemJoin.ilk`, `gemJoin.gem` and `usddJoin.usdd` could be declared as immutables.
- `RwaLiquidationOracle`: `init`, `bump`, `cull`, `cure` and `good` perform several extra storage reads of the variable `ilks[ilk]`.
- `RwaSwapOutConduit`: `_doPush` performs one extra storage read of the variable `to`. Similarly, `_doQuit` performs a duplicate read to `quitTo`.

---

**Acknowledged:**

Decentralized USD acknowledges the issue.

## 7.3 Inaccurate Documentation

Informational | Version 1 | Code Partially Corrected | Acknowledged

*CS-USDD-RWA-007*

The documentation outlines a set of roles and the actions they are allowed to perform. However, some descriptions do not match the implementation.

**Confusing Roles.** The operator and the user roles are seemingly confused either by the naming of variables, the documentation or the implementation.

In particular, the role "User" is described as follows:

```
Sets the recipient address for funds. Applies to: RwaSwapOutputConduit: pick
```

While the "Operator2" role is described as:

```
Has the authority to initiate token transfers. Applies to: RwaSwapOutputConduit: push, quit [...]
```

However, the `RwaSpell` gives `RWA001_OPERATOR` (similar name to role "Operator2") the permission to call `push`. However, it does **not** give `RWA001_USER` (similar name to role "User") the permission to `pick` but whitelists the address to receive the funds:

```
RwaOutputConduitLike(RWA001_A_OUTPUT_CONDUIT).kiss(RWA001_USER); // who can pick to set to address
```

Additionally, the inline comment hints that `RWA001_USER` should be able to pick the receiving address, which is not the case.

Ultimately, the operator will call `pick` while the user will be a whitelisted destination for funds.

**Inaccuracies.** The documentation specifies that the authority to call `RwaUrn2.quit` is anyone, suggesting that the function is permissionless. However, that is only the case when the Vat is not live and the shutdown has been initiated. During regular operations, the role "Operator1" has authority over the function.

---

**Code partially corrected:**

Decentralized USD has changed the inline comment in `RwaSpell_mainnet` to clarify that `RWA001_USER` is the whitelisted recipient of the funds, and **not** the user able to pick the receiving address.

**Acknowledged:**

The documentation still suggests that `RwaUrn2.quit` can be called by anyone, which is not the case during regular operations.

## 7.4 NatSpec Mismatches and Inaccuracies

Informational | Version 1 | Code Partially Corrected | Acknowledged

*CS-USDD-RWA-008*

NatSpec may inform users with descriptive comments about the functionality of contracts.

Some NatSpec is inaccurate. Below is a non-exhaustive list of NatSpec mismatches and inaccuracies:

- `RwaTokenFactory.RwaTokenCreated`: The event's NatSpec lacks the `token` parameter of the event. The `recipient` is documented as `Token address recipient` which is vague.
- `RwaUrn2.constructor`: The parameters are commented in the wrong order (e.g. the jug is described as the gem join).

- `RwaJar.constructor`: The `vow_` parameter remains undocumented.

Some contracts are missing NatSpec comments. Below is a non-exhaustive list of contracts missing NatSpec documentation:

- `RwaToken`
- `DSValue` and `DSAuth`
- `RwaLiquidationOracle`
- `AuthGemJoin`

---

**Code partially corrected:**

Decentralized USD has addressed the inaccuracies in `RwaTokenFactory`, `RwaUrn2`, and `RwaJar`.

**Acknowledged:**

Decentralized USD acknowledges that NatSpec comments are still missing from the following contracts: `RwaToken`, `DSValue`, `DSAuth`, `RwaLiquidationOracle`, and `AuthGemJoin`.

# 8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

## 8.1 Bypassing Stability Fee

Note Version 1

The `RwaUrn2.wipe` function suggests that the rate will be updated prior to wiping the debt:

```
jug.drip(ilk);
...
vat.frob(ilk, address(this), address(this), address(this), 0, -int256(dart));
```

However, governance should be aware that `jug.drip` should be called frequently if the rate corresponds to interest greater than 0% since the call to `drip`, enforced by the RWA urns `wipe` function, can be trivially bypassed by operators (or any other address) by calling `Vat.frob` directly to wipe debt.

## 8.2 Init Spell References Chainlog

Note Version 1

The documentation specifies that no Chainlog contract is present in the system:

```
The USDD system currently does not include the chainlog contract [...]
```

However, the init spell `RwaSpell` references the Chainlog to publish the addresses with `setAddress`. Additionally, Decentralized USD specified that the Chainlog will be deployed prior to the execution of the spell. It is important to note that if Chainlog is not deployed prior to spell execution, the spell could revert

## 8.3 Spells Are Templates

Note Version 1

Governance should be aware that the spells need to be adjusted prior to deployment to set the expected parameters. Otherwise, incorrect default parameters could be configured for RWAs.

## 8.4 Vow Is Immutable in RWA Jar

Note Version 1

Governance and users should be aware that the `vow` in `RwaJar` is immutable. In contrast, the liquidation oracle allows for updating the `vow` with `file`.

In case the Vow is replaced, the RWA jar should be redeployed to reflect the change.