PUBLIC

# Code Assessment

## of the PSM Smart Contracts

January 24, 2025

Produced for

by

CHAINSECURITY

# Contents

# 1 Executive Summary

Dear Decentralized USD team,

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 PSM according to Scope to support you in forming an opinion on their security risks.

Decentralized USD implements a Peg Stability module (PSM) for USDD V2. The PSM is a system designed to help maintain the peg of USDD by enabling the direct exchange of USDD for supported stablecoins (and vice versa) at a fixed exchange rate of 1:1.

The most critical subjects covered in our audit are asset solvency, functional correctness, and access control. The reported issue Incorrect USDT Address has been resolved, hence security regarding all the aforementioned subjects is high.

The general subjects covered are gas optimizations and specification. Security regarding both subjects is high.

In summary, we find that the codebase provides an good 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 | 1 |
| • Code Corrected | 1 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 0 |
| Low-Severity Findings | 0 |

# 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 PSM 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 | 16 Dec 2024 | 9a28413b5bc354e182eb48fea472bc385b981afc | Initial Version |
| 2 | 24 Jan 2025 | c368445876334664d6e58b7e186c78162271ff93 | After Intermediate Report |

For the solidity smart contracts, the compiler version `0.6.12` was chosen.

The following files were in the scope of this review:

```
src/
    join-5-auth.sol
    join-8-auth.sol
    join-auth.sol
    psm.sol
```

### 2.1.1 Excluded from scope

Any other file not explicitly mentioned in the scope section is considered out of scope. In particular, the tests and external dependencies are not part of this audit.

In addition, the following known risks are out of the scope of this review:

- Holding a significant amount of centralized tokens in the PSM, like USDT, carries inherent risks of centralization.
- Repeatedly swapping Gem for USDD might efficiently bloat the global debt to Line and block borrowing with other collaterals.

## 2.2 System Overview

This system overview describes the initially received version (Version 1) 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 a Peg Stability module (PSM) for USDD V2. The PSM is a system designed to help maintain the peg of USDD by enabling the direct exchange of USDD for supported stablecoins (and

vice versa) at a fixed exchange rate of 1:1. The system allows users to trade stablecoins into USDD (`sellGem`) or trade USDD into stablecoins (`buyGem`).

This implementation is based on a fork of the MakerDAO PSM contract, with modifications to provide additional operational control. The PSM interacts with a fork of the VAT, a core MakerDAO accounting system.

The system is composed of two main types of contracts:

1. The PSM contract: `UsddPsm`
2. The Join contracts: `AuthGemJoin`, `AuthGemJoin5`, `AuthGemJoin8`

### 2.2.1 PSM Contract

The UsddPsm contract is functionally similar to the MakerDAO PSM contract, with the primary addition of two boolean parameters:

- `sellEnabled`: Enables or disables the `sellGem` function.
- `buyEnabled`: Enables or disables the `buyGem` function.

The contract provides two main functions:

- `sellGem(address, uint256)`: Converts the stablecoin `gem` into USDD. When invoked, the stablecoin is deposited into the associated Join contract and accounted by the VAT, and the user receives USDD. A fee defined by the `tin` parameter is charged and credited to the VOW.
- `buyGem(address, uint256)`: Converts USDD into the stablecoin `gem`. When invoked, USDD is exited to the user by the USDD Join after the VAT updates the internal accounting. A fee defined by the `tout` parameter is charged and credited to the VOW.

The contract also includes administrative functions, which are protected by the `auth` modifier. The `auth` modifier restricts access to `wards`, that is, authorized addresses. These administrative functions include:

- the `rely` and `deny` functions to add or remove a `ward`,
- the `file` function to change the parameters of the PSM (`tin`, `tout`, `sellEnabled`, `buyEnabled`),
- the `hope` and `nope` functions to enable or disable the ability of another address to interact with the VAT on behalf of the contract.

### 2.2.2 Join Contracts

Join contracts are implemented to faciliate the interaction between the PSM and different stablecoins. They handle deposits and withdrawals of stablecoins while managing their corresponding balances in the VAT.

Each Join contract provides two primary functions:

- `join(address, uint256, address)`: Deposits a specific stablecoin. The stablecoin is transferred to the Join contract, which then increase the ilk balance of a specific urn in the VAT with `slip()`. Note this function is restricted to the `wards`.
- `exit(address, uint256)`: Withdraws a specific stablecoin. The Join decreases the ilk balance of a specific urn in the VAT with `slip()`, and then transfer the `gem` token to the user.

And some administrative functions:

- `rely` and `deny` functions to add or remove a ward.

- `cage` function to disable the join and exit functions.

There are three types of Join contracts:

- `AuthGemJoin`: Basic token adapter for tokens with 18 decimals.
- `AuthGemJoin5`: Adapter for stablecoins with fewer than 18 decimals of precision (e.g., USDC with 6 decimals).
- `AuthGemJoin8`: Adapter for upgradable stablecoins with fewer than 18 decimals (e.g., GUSD with 2 decimals). An extra function `setImplementation` is added to set the permitted implementations for this token.

### 2.2.3 Roles & Trust Model

It is assumed that the wards (i.e., administrators) in the system are configured as follows for each contract:

- PSM Contracts: Only the `DsPauseProxy` is granted ward privilege to manage administrative functions.
- Join Contracts: Both the `DsPauseProxy` and their corresponding PSM contract are granted ward privilege, allowing the PSM to call `join()`.

The `DsPauseProxy` is a governance contract within USDD v2 system that holds elevated permissions to execute administrative functions across the system. It is considered fully trusted within the scope of this audit.

The following assumptions are further made for the `ilk` of PSM:

- There are no other urns for the same ilk.
- Stability fee is always zero for the ilk (ilk.rate==RAY).
- The spot price for gem is always 1 (ilk.spot==RAY).
- No liquidations on this specific ilk.

Weird tokens such as rebasing tokens and tokens with transfer fees are not expected to be used. The system is also subject to the potential risks of upgradability, blacklisting, pausing, and frozen of the gem tokens.

# 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 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:

- 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 | 0 |

# 6 Resolved Findings

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

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

| | |
|---|---|
| Critical-Severity Findings | 1 |

- Incorrect USDT Address Code Corrected

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

## 6.1 Incorrect USDT Address

Correctness Critical Version 1 Code Corrected

*CS-USDD-PSM-001*

`AuthGemJoin5` hardcoded the USDT token address (`USDTAddr`) as a constant in order to execute tailored logic for handling USDT transfers. However, the hardcoded address does not point to the USDT address on Tron mainnet.

As there is no contract published at this address on mainnet, the low level calls to this address will always succeed. Hence the call of `safeTransfer()` and `safeTransferFrom()` will both silently succeed. Consequently, anyone can come to the PSM related to this Join and call `sellGem()` to print USDD for free.

---

**Code corrected:**

The hardcoded address has been replaced with the correct address of USDT on mainnet.

# 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 Hardcoded Function Selector

Informational Version 1 Acknowledged

*CS-USDD-PSM-002*

The `safeTransfer` and `safeTransferFrom` functions use hardcoded function selectors (`0xa9059cbb` and `0x23b872dd`) for the `transfer` and `transferFrom` functions, respectively, which reduces readability and makes the code less intuitive. Replacing these hardcoded values with interfaces or selectors derived from the function signatures would improve clarity and maintainability.

---

**Acknowledged:**

Decentralized USD has acknowledged this issue and decided not to change it.

## 7.2 Inconsistent Coding Style

Informational Version 1 Acknowledged

*CS-USDD-PSM-003*

The `file` function in `psm.sol` uses inconsistent braces for conditional branches. Some branches (e.g., `tin`, `tout`) omit braces for single statements, while others (e.g., `sellEnabled`, `buyEnabled`) include them.

---

**Acknowledged:**

Decentralized USD has acknowledged the inconsistent coding style and decided not to change it.

# 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 Proper Configuration of PSM, Join, and Ilk

Note Version 1

Proper configuration is critical to ensure the PSM operates as expected. Each PSM must be paired with its specific Join contract, and the Join contract must grant the PSM ward access. Additionally, the PSM must be associated with a dedicated Ilk in the VAT. The Ilk must be configured with appropriate risk parameters, such as the liquidation ratio, the stability fee, and the debt ceiling.