PUBLIC

Code Assessment

of the PSM Lite Smart Contracts

July 04, 2024

Produced for

by

Contents

1 Executive Summary

Dear all,

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

MakerDAO implements a gas-efficient Peg Stability Module (PSM) where users can freely swap Dai for stablecoins. Further, a phased migration from the old PSM to the new PSM is implemented.

The most critical subjects covered in our audit are functional correctness, assets solvency and the correct adherence to the MakerDAO specifications. Security regarding all the aforementioned subjects is high.

The general subjects covered are access control, interaction with third party systems and the documentation. Security regarding all the 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	5
• Code Corrected	3
• Specification Changed	1
• Risk Accepted	1

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 Lite repository based on the documentation files.

The scope consists of the two solidity smart contracts:

1. ./src/DssLitePsm.sol
2. ./src/DssPocket.sol

In version 4 the following files have been added:

1. script/DssLitePsmDeploy.s.sol
2. script/dependencies/DssLitePsmDeploy.sol
3. script/dependencies/DssLitePsmInit.sol
4. script/dependencies/DssLitePsmInstance.sol

In version 5, ./src/DssLitePsmMom.sol has been added.

In version 8, ./src/DssPocket.sol has been removed.

In version 10, the deployment process has been redesigned and divided into three phases. Files have been moved and added; the following files are in scope:

1. deployment/phase-1/DssLitePsmMigrationPhase1.sol
2. deployment/phase-2/DssLitePsmMigrationPhase2.sol
3. deployment/phase-3/DssLitePsmMigrationPhase3.sol
4. deployment/DssLitePsmInstance.sol
5. deployment/DssLitePsmDeploy.sol
6. deployment/DssLitePsmMigration.sol
7. deployment/DssLitePsmInit.sol

In this review, it is assumed that the deployment and initialization scripts are explicitly used to deploy a new LitePSM-USDC and migrate the existing PSM to the new one.

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	16 October 2023	5a1d92d53a3cd6958073faf534b57b8eb50e3882	Initial Version
2	19 October 2023	3ec57f35fdd910ab765379c324a4dc2a7c08d54a	Updated Design
3	20 October 2023	bc6f46d4a9b254f43f009c339f547e401208403b	After Intermediate Report
4	13 November 2023	1ee98ee06ba27ee35166f5b62de19d7abe617916	Deployment Script

5	22 November 2023	7d3af40cb539936bfe690e6d96c95996cf648660	PSM Mom
6	23 November 2023	aadc4b2909fb4feb1b9998d137aede5f57c73169	Fix Ilk Class
7	05 January 2024	05ddf065a6c0242e0ca497bf93defde899c2e2fc	Refactor Halt Swaps
8	04 March 2024	b91c6fecabfda91cfa394132b3b58043f2e4ea3b	Remove DssPocket
9	29 April 2024	374bb08b09a3f4798858fd841bab8e79719266c8	Interface Compatability
10	24 June 2024	1f12ceda78ab85e926c4161164aa2f9531756c7d	New Deployment Scripts
11	03 July 2024	2f11f4bc47d96f4ebf025a1e4a249987150f9baa	Hardcoded *dstWant*

For the Solidity smart contracts, the compiler version `0.8.16` was chosen.

2.1.1 Excluded from scope

Any other files not explicitly mentioned in the scope section. In particular tests, scripts, external dependencies, and configuration files are not part of the audit scope.

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 USDC, carries inherent risks of centralization.
- Similar to the existing PSM implementation, repeatedly swapping Gem for Dai 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.

At the end of this report section we have added subsections for each of the changes accordingly to the versions.

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

MakerDAO offers Peg Stability Module (PSM) Lite, a gas-efficient facility through which users can freely swap Dai for stablecoins with no slippage.

2.2.1 DssLitePsm

A PSM enables users to swap between Dai and a specific stablecoin at a fixed 1:1 conversion rate (with a fee if it is turned on). PSM-Lite is more gas efficient compared to its predecessors due to the separation of the swaps and VAT manipulations. Users can directly swap in a pool of pre-minted Dai and stablecoins without interacting with the VAT. The pool can be refilled or deflated up to certain limits in a permissionless way, which involves VAT interactions. The Dais are minted using a special ilk in the VAT, the actual gem tokens are held by the pocket contract.

The contract is governed by `wards` with the privilege to the following functions:

- `rely()` - grants admin privilege to an address.

- `deny()` - revokes admin privilege from an address.
- `kiss()` - grants an address the permission to swap without fees.
- `diss()` - revokes the permission of swapping without fees from an address.
- `file(bytes32 what, address data)` - update the `VOW` address.
- `file(bytes32 what, uint256 data)` - update the fee-related parameters `tin` and `tout` and buffer size `buf`.

The following public functions are provided:

- `rush()` - returns the missing Dai that can be filled into this contract. The target is to have `buf` amount of DAI available. Minting DAI is limited by the ilk limit (`line`) and the global debt limit (`Line`).
- `gush()` - returns the excess Dai that can be trimmed from this contract. The target is to leave `buf` amount of DAI at the contract, surplus is to be trimmed. In case the ilk's debt limit (`line`) is currently exceeded (as it was reduced), the function attempts to return all DAI necessary. In version 1 of the code `gush()` additionally takes `Line` into account. The amount of DAI returned is capped by the currently available DAI balance.
- `cut()` - returns the amount of accumulated swapping fees that can be harvested from this contract.

It implements the following permissionless entry points:

- `sellGem()` - a user can swap gem tokens to pre-minted Dai with a fee. The gem tokens will be sent to the pocket contract.
- `buyGem()` - a user can swap Dai tokens to gem with a fee. The gem tokens will be transferred from the pocket contract to the receiver.
- `fill()` - a user can mint more Dai to the pool by the amount of `rush()`.
- `trim()` - a user can trim Dai from the pool by the amount of `gush()`.
- `chug()` - a user can push the accumulated fees to the `VOW`. The Dai tokens are deposited into DAIJoin with the VOW as beneficiary.

Besides, `buyGemNoFee()` and `sellGemNoFee()` are the permissioned counterparties for `buyGem()` and `sellGem()`, designed for the authorized parties to swap without fees.

2.2.2 DssPocket

The gem received from the swaps will be placed in the pocket contract, which is fully governed by its wards with the following permissioned entry points:

- `rely()` - grants admin privilege to an address.
- `deny()` - revokes admin privilege from an address.
- `hope()` - grants an address the permission to spend `max(uint256)` gem on behalf of this contract.
- `nope()` - fully revokes an address's permission to spend gem on behalf of this contract.

2.2.3 Changes in Version 2

- Instead of creating/burning the special ilk to increase its `gem` balance every time in `fill()` and `trim()`, the `urn` for DssLitePsm in the `VAT` is expected to have "unlimited" `ink`. DssLitePsm no longer manipulates the `ink` of the urn.
- The excess Dai that can be trimmed from this contract (`gush()`) has been changed to disregard the gap between the global debt (`debt`) and limit (`Line`).

2.2.4 Changes in Version 4

A deployment script has been added in Version 4, which deploys a DssLitePsm with a pocket and migrates the existing PSM (SrcPsm) to the new DssLitePsm.

The following sanity checks are adopted in `DssLitePsmInit`:

- The key used for srcPsmKey, DssLitePsm, and pocket are different in chainlog.
- The newly deployed DssLitePsm and pocket are connected correctly.
- SrcPsm has the same gem as DssLitePsm
- The price of SrcPsm's ilk is 1.
- SrcPsm's ilk is only used by SrcPsm, and SrcPsm's ink equals SrcPsm's art.

Then, the following steps are executed for the initialization and migration:

1. The specific ilk for DssLitePsm is added to VAT, JUG and SPOT. The pip of SrcPsm is reused and the price is set to 1.
2. The individual line of the specific ilk and Line are updated by art of SrcPsm (`src.art`) in VAT for DssLitePsm.
3. `max(uint256) / RAY` amount of specific ilk is minted to DssLitePsm and locked into DssLitePsm's urn.
4. The `buf` of DssLitePsm is set to `src.art`, fees are set to 0, and it is filled with pre-minted Dai (`DssLitePsm.fill()`). This is done to allow gem migration.
5. Migrate the gem from SrcPsm:
    1. Grab the collateral and debt from SrcPsm to the initializing contract.
    2. Exit the gem tokens from GemJoin and transfer to the initializing contract.
    3. Swap the gem tokens to Dai in DssLitePsm.
    4. Erase the bad debt by the swapped Dai via `vat.heal()`.
6. Set the auto-line configuration for DssLitePsm to enable adjusting the line of the specific ilk in a permissionless way.
7. Set actual parameters of DssLitePsm (`buf`, `tin`, and `tout`).
8. Fill the DssLitePsm to make liquidity available immediately after initialization.
9. Add the specific ilk to the ilk registry.
10. Add DssLitePsm and pocket to chainlog.

2.2.5 Changes in Version 5

DssLitePsmMom has been added to the codebase in Version 5, it is designed to execute governance actions on the PSM without delay. The governance can halt (`halt()`) either the inflow or the outflow of the gems from the PSM by setting the swap fees to a special value (`type(uint256).max`).

DssLitePsm and the deployment scripts have been adapted to reflect the changes.

2.2.6 Changes in Version 7

Functionality to halt the inflow or the outflow of the gems via `sellGemNoFee()` and `buyGemNoFee()` has been added to the codebase in Version 7. Now both functions will behave the same as their permissionless couterparties if the swap fee has been set to `type(uint256).max` (`HALTED`).

2.2.7 Changes in Version 8

DssPocket has been removed in Version 8. The deployment and initialization scripts have been adjusted: the DssPocket is no longer deployed in the deployment script, and will be passed as a parameter in both scripts. It is checked in the initialization script that the pocket grants `type(uint256).max` allowance to the DssLitePsm.

2.2.8 Changes in Version 9

The DssLitePsm is intended to act as a drop-in replacement for the current PSM in the RWA conduits. To ensure compatibility, the `gemJoin()` function returning `address(this)` has been implemented. In addition, the DssLitePsm implements the view functions `dec()` and `live()` so that the contract supports all the view functions defined in the `join-5-auth` interface (to prevent potential compatibility issues in external systems).

- `dec()`: returning the decimals of the DssLitePsm gem.
- `live()`: returning whether the VAT is live. Note this only reflects if the VAT is caged. The DssLitePsm is halted when the swap fees are set to a constant `type(uint256).max`.

2.2.9 Changes in Version 10

The deployment process has been redesigned and divided into three phases as outlined in the forum post (archive):

- Phase 1 - Test period
- Phase 2 - Main migration
- Phase 3 - Final migration

Phase 1:

Based on the provided configuration argument, the LitePsm is initialized and an initial portion of funds is migrated to the new Psm. The AutoLine configuration for both Psms is updated, the final buffer (`buf`) for the new Psm is set, and the Psm is filled if necessary.

After this phase, both Psms are active, with fees (`tin`, `tout`) remaining unchanged.

Phase 2:

Based on the provided configuration argument, the migration library is used to transfer funds from the source Psm to the new LitePsm (the destination Psm). It is ensured that the source Psm's ink is at least `srcKeep` after the operation. The AutoLine configuration for both Psms is updated, the final buffer (`buf`) for the new Psm is set, and the Psm is filled if necessary.

After this phase, both Psms are active. No fees (`tin`, `tout`) have been set on the new PSM, but non-zero fees have been set on the source PSM to discourage interactions that could disrupt the migration.

Phase 3:

Using the migration library, the remaining funds are transferred to the new Psm. The old Psm (srcPsm) is then offboarded from the system:

- The corresponding `ilk` is removed from AutoLine.
- The `line` of this `ilk` is set to 0, and the global `Line` is reduced accordingly.
- The fees for this Psm are reset to 0.

The buffer of the new Psm is set.

With the migration completed, the old Psm has been offboarded and is now inactive, with all funds moved to the new LitePsm.

Initialization and Migration

The initialization in Phase 1 (implemented in the adjusted `DssLitePsmInit`) is similar to the initialization steps in the previous version (see Changes in Version 4). Namely, step 1., 3., 9. and 10. are performed. Note that the mom is configured accordingly (see Changes in Version 5). Also note that the VOW is set in the LitePsm and that the `MCD_PAUSE_PROXY` is given the `bud` role in the LitePsm, to swap with no fees. Sanity checks are implemented similarly as in Changes in Version 4.

The spells for the different phases share the logic of the migration in `DssLitePsmMigration` which implements the migration of the funds. Note that this is similar to the previous version (see Changes in Version 4 point 5.). However, the main differences are

1. that the debt ceilings `Line` and `line` are temporarily raised to the maximum (and similar changes to allow separation of migration logic)
2. and that the migrated amounts differ. Namely, the migration tries to reach `srcKeep` but is allowed to migrate at most `dstWant`. Note that checks are implemented in phase 2 to ensure `srcKeep` is respected.

Note that the sanity checks for the migration are similar to the migration-related one in Changes in Version 4.

2.2.10 Roles and Trust Model

The DS-Pause-Proxy is expected to be the ultimate admin (ward role) of the DssLitePsm. The role should be set after the deployment by the deployer. Any privileged action performed on the contracts, is expected to be executed through well-considered and inspected governance spells. In case there are any other wards of the contracts, they are assumed to be fully trusted and never act against the interest of the system and users.

The state variables `VOW`, `tin`, `tout`, `buf`, `bud` of DssLitePsm are not initialized in the constructor. They should be properly set upon deployment, otherwise the contract cannot function correctly. For example, in case `buf<ilk.dust` or `buf>ilk.line`, no Dai can be pre-minted due to the checks in VAT.

The wards of DssPocket are trusted. It is assumed that only the DssLitePsm has `max(uint256)` allowance from DssPocket, which can freely transfer `gem` on behalf of pocket. The bud of DssLitePsm, who can swap without fees, are fully trusted. The users of DssLitePsm are not trusted.

The following assumptions are further made:

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

The gem tokens used are assumed to be within 18 decimals. 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.

For the deployment script in (Version 4), it is assumed that the `JUG.base` will be 0, so there would be no interest accrued for the new PSM. The deployers are supposed to deploy the contracts as specified by the reviewed script. Deployers are, however, EOAs that could perform unlawful actions besides simple deployment, such as changing the settings of the newly deployed or granting themselves special privileges. This cannot be fully inspected through the initialization library, which is expected to be executed via governance spells. It is important that after deployment, concerned parties thoroughly check the state of the deployed contracts to ensure that no unexpected action has been taken on them during deployment (deployment validation).

In (Version 5), it is further assumed the PSM Mom has the ward role on DssLitePsm. The owner of PSM Mom is assumed to be the DS-Pause-Proxy, and its authority is assumed to be the `MCD_ADM`.

In Version 8, the DssPocket has been removed, its actual implementation and access control will be unknown. This implies:

- The implementation and access control of DssPocket will be unknown.
- The DssPocket may not be controlled by the DS-Pause-Proxy anymore.
- It's assumed that it issues an approval on gem for the LitePsm instance with type(uint256).max as the amount.

Hence the DssPocket should be fully trusted, otherwise the DssLitePsm is exposed to the potential loss of gem tokens.

In Version 9, three view functions have been added to comply with the view functions defined in `join-5-auth` for RWA conduits integration. Note that some external functions (e.g. `cage()`, `join()`, `exit()`) have not been added, hence any potential calls to them will revert. It is assumed that the external systems fully understand the different semantics between DssLitePsm and the current PSM with a GemJoin, and will use the view functions with caution.

In Version 10, the migration has been changed. While the previous assumptions on trust remain, we further expect that the migrated amounts will be high enough so that the fees to be paid in case of manipulations will be non-negligible. Further, we expect that the amounts are high enough so that a swap on DEXs is not realistically possible (fees and slippage are too high). We expect the phases to be executed in order. Further, we expect, as outlined in the forum post specifying the migration, that the GSM delay will be reduced and that the ESM threshold will be increased in other scripts. Last, note that phase 1 for example has no guarantees about the execution (given that no checks are done as in phase 2). Also, note that if bad parameters are chosen for phase 2, the migration could result in a non-optimal state.

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:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- 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	1

- Unchecked Return Value of transferFrom Risk Accepted

5.1 Unchecked Return Value of transferFrom

Security Low Version 1 Risk Accepted

CS-MKPSML-001

In DssLitePsm, the return value of `transferFrom()` is not checked. It relies on the token to revert if the transfer fails which is given for the intended gem token (USDC). Generally however, according to the ERC20 specification upon a failed transfer the token may revert or simply returns `false`. In case the gem token's implementation returns false on a failed `transferFrom()` instead of reverting, DssLitePsm will still treat it as a success and proceed.

Risk accepted:

MakerDAO states:

```
We do not plan to support ERC-20 tokens that do not revert on a failed transfer.
We might want to support tokens that do not return true on succeeded transfers.
```

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	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	4

- Phase 2 Might Not Migrate Gem Code Corrected
- Incorrect Ilk Class Code Corrected
- Uninitialized Vow in DssLitePsmInit Code Corrected
- Inaccurate Specification Specification Changed

Informational Findings	3

- Fill More Than buf Possible Specification Changed
- Unpermissioned Repay of Art, Remaining Collateral Balance Code Corrected
- The Optimizer Is Disabled Code Corrected

6.1 Phase 2 Might Not Migrate Gem

Design Low Version 10 Code Corrected

CS-MKPSML-010

The goal of phase 2 of the migration is to migrate a bigger portion of the funds to the PSMLite. Checks and fees mainly disincentivize or prevent manipulations. However, there remains the possibility to "undo" a migration for free (ultimately, no *gem* is migrated).

Consider the following order of operations:

1. The `ink` of the `srcPSM` is defined as `srcKeep + x`
2. `srcPSM.sellGem dstWant` so that the new `ink` is `srcKeep + x + dstWant`.
3. Cast spell for phase 2. Consequently, `dstWant` is migrated to the `dstPSM`. The `ink` of `srcPsm` is again `srcKeep + x`.
4. `dstPSM.buyGem` to get `dstWant` out and repay a hypotethetical flashloan.
5. As a consequence, the `srcPSM` again has ``

Note that similarly the order of casting a spell and `buyGem` might be reordered (however, would limit the manipulation more).

Ultimately, while the migration might seem successful, the spell might not effectively migrate funds from the old PSM to the new PSM.

Code corrected:

MakerDAO intended to use the maximum unsigned integer for `dstWant`. The value is now hardcoded to ensure correct execution.

6.2 Incorrect Ilk Class

Correctness | Low | Version 4 | Code Corrected

CS-MKPSML-006

Regarding `ilk.class`, the Ilk Registry states:

```
Classification code (1 - clip, 2 - flip, 3+ - other)
```

In PSM initialization script (`DssLitePsmInit.sol`), the class of the existing PSM (`src.class`) is reused to register the specific ilk for DssLitePsm (For PSM-USDC, `src.class==1`). Nevertheless, in contrast to the existing PSM, DssLitePsm does not have auction modules and cannot be liquidated. Hence directly reusing `src.class` does not comply to the specifications.

Code corrected:

The `ilk.class` for the newly deployed DssLitePsm has been changed to a constant 6 (a new IlkRegistry class), representing a specific ilk type without an associated GemJoin.

6.3 Uninitialized Vow in DssLitePsmInit

Correctness | Low | Version 4 | Code Corrected

CS-MKPSML-009

Library `DssLitePsmInit` does not initialize the vow of the newly deployed DssLitePsm. Consequently the outstanding accumulated fees cannot be harvested into the surplus buffer (`chug()`), and it would require another governance spell to set the vow.

Code corrected:

MakerDAO has corrected the code in Version 5. The vow is now initialized in the `DssLitePsmInit.init` function:

```
DssLitePsmLike(inst.litePsm).file("vow", dss.chainlog.getAddress("MCD_VOW"));
```

6.4 Inaccurate Specification

Correctness | Low | Version 1 | Specification Changed

CS-MKPSML-007

The specification of DssPocket states "Can grant or revoke infinite *gem* approvals". Not all gem tokens treat `max(uint256)` as truly unlimited allowance, though it is unlikely to exhaust `max(uint256)` allowance in practice .

Specification changed:

The specification has been updated to read up to max(uint256) instead of infinite.

6.5 Fill More Than buf Possible

Informational | Version 1 | Specification Changed

CS-MKPSML-008

The specification of fill() states "Mints Dai into this contract up to buf value". This is not entirely accurate since the amount of DAI to be minted depends on the current art of the urn and the amount of gem at the pocket. Both of which can be manipulated by anyone if one is ready to spend money to:

- Inflate tArt by donating gem tokens to DssPocket.
- Deflate Art by repaying on behalf of DssLitePsm in VAT.

Consequently the amount to be refilled (computed in rush()), depending on the actual circumstances may result in:

- More than buf amount of DAI at the DssLitePsm after fill().
- More than buf amount of DAI minted during fill().

Specification changed:

The specification has been updated to reflect how the extraneous influences might affect the amount minted by fill().

6.6 The Optimizer Is Disabled

Informational | Version 1 | Code Corrected

CS-MKPSML-012

The optimizer is disabled in foundry.toml. Enabling the optimizer can help to further improve the gas efficiency.

Code corrected:

The optimizer has been enabled in foundry.toml.

6.7 Unpermissioned Repay of Art, Remaining Collateral Balance

Informational | Version 1 | Code Corrected

CS-MKPSML-011

In the VAT it's possible to repay debt for any urn by using VAT.frob() without requiring permissions. If this is done for the special urn of DssLitePsm, this has the following consequences:

- DssLitePSM's urn art will reduce while ink remains unchanged.
- Once the fees are collected through chug() the corresponding surplus will be paid out as part of the fees. The collateral accounting in the VAT is not updated, the ink balance remains, despite the funds now having left the system.

Code corrected:

In Version 2 the design has been changed, the `ink` of the urn is expected to be unlimited and is no longer modified by this contract.

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 Lack of Sanity Checks

Informational **Version 4**

CS-MKPSML-002

While the `DssLitePsmInit` script implements many sanity checks, `DssLitePsm.to18ConversionFactor()` is not validated to be the expected value.

7.2 No Dai May Be Available if Limit Is Reached

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

CS-MKPSML-003

DssLitePsm may not achieve its full functionalities in case the individual debt limit (`ilk.line`) or the global debt limit (`VAT.Line`) is reached. In this case, the pool cannot be refilled. As long as no or insufficient DAI balance is available, users can only swap DAI for gem tokens (which in turn makes some DAI available).

Acknowledged:

MakerDAO has acknowledged the behavior of the pool if the debt limit is reached.

7.3 No Fees on Small Swap Amounts

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

CS-MKPSML-004

Following the design of the existing PSMs, the fees of the swaps are computed and rounded down as:

```
fee = daiOutWad * tin_ / WAD

fee = daiInWad * tout_ / WAD;
```

As a result, amounts of `daiOutWad` or `daiInWad` may avoid the swap fees due to the rounding errors. Compared to the gas costs, this is negligible.

Acknowledged:

This behavior is now documented in the code.

7.4 Trim May Fail if Art Is Below Dust

Informational | Version 1 | Acknowledged

CS-MKPSML-005

`Trim()` will burn Dai and decrease the debt `urn.art`. If the resulting `art` of the urn is non-zero but below `ilk.dust` this fails. For this special collateral, `ilk.dust` should be set to 0.

Acknowledged:

MakerDAO has acknowledged the edge case if `ilk.dust` is not set properly.

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 Changes Compared to the Existing PSMs

Note Version 1

There are currently three active PSMs for USDC, Gemini-USD and Pax-USD at the time of this review (October 2023). The PSM Lite has the following changes:

1. In the existing PSMs the gem tokens are deposited into a Gemjoin adapter before `art` is generated and DAIs are minted. PSM Lite use a special ilk for Dai minting ahead of time. Once these DAIs are exchanged for gem tokens, these tokens are held within DssPocket.
2. The volume for a single swap is only limited by the `ilk.line`, `debt`, and `Line` in the existing implementation. PSM Lite introduces a pool of pre-minted Dai limited by `buf` size, which may not satisfy a single swap with an exceeding volume. In this case the single swap should be broken down into several smaller swaps, in between of which the pool can be refilled (`fill()`) by the user.

8.2 Considerations for Initialization

Note Version 1

While the initialization script for the PSM performs many sanity checks to prevent malicious deployments, it cannot check everything. The governance should, before approving any spell, carefully evaluate the deployment process. Namely, the following should be considered:

1. The init code of the contracts should match the init code generated by the compiler with the correct immutables attached. As a consequence, it can be ensured that the bytecode is correct and that the constructor has not been altered to perform malicious actions (e.g. give approvals to arbitrary addresses).
2. The only successful calls made to the contracts after their creation should be the following (in the same order for both contracts):
    1. `rely()` to give the governance a privileged role.
    2. `deny()` to remove the privileged role from the deployer.

8.3 DssLitePsm Does Not Support Global Settlement

Note Version 1

At the time of this review, Emergency Shutdown Module (ESM) is still functioning and could be triggered by users burning sufficient MKR. However, DssLitePsm does not support the coordinated Shutdown / Global Settlement anymore. It is expected that the ESM is disabled by setting its threshold large enough prior to the deployment of `DssLitePsm`, so Emergency Shutdown can never be called.

8.4 Swaps Are Subject to Front-Running

Note Version 1

A user's swap can be front-run by another swap of the same direction, which will leave insufficient Dai or gem tokens and revert the user's swap. In the former case, users can bundle a `fill()` before the swap to refill the pool with newly minted Dai. This is well documented in the README of PSM Lite.