cecuro

Audit Report

March 4, 2026

PROJECT

Sky Money

Audit Overview

Project:	Sky Money
Repository:	https://github.com/sky-ecosystem/dss-lite-psm
Audit Date:	March 4, 2026
Commit:	2598e2ef
Scope:	16 files

0	0	0	17	48
CRITICAL	HIGH	MEDIUM	LOW	INFO

Audit Scope

The following 16 files were included in this security audit:

- src/DssLitePsm.sol
- src/DssLitePsm.t.integration.sol
- src/DssLitePsmMom.sol
- src/DssLitePsmMom.t.integration.sol
- src/deployment/DssLitePsmDeploy.sol
- src/deployment/DssLitePsmInit.sol
- src/deployment/DssLitePsmInit.t.integration.sol
- src/deployment/DssLitePsmInstance.sol
- src/deployment/DssLitePsmMigration.sol
- src/deployment/DssLitePsmMigration.t.integration.sol
- src/deployment/phase-1/DssLitePsmMigrationPhase1.sol
- src/deployment/phase-1/DssLitePsmMigrationPhase1.t.integration.sol
- src/deployment/phase-2/DssLitePsmMigrationPhase2.sol
- src/deployment/phase-2/DssLitePsmMigrationPhase2.t.integration.sol
- src/deployment/phase-3/DssLitePsmMigrationPhase3.sol
- src/deployment/phase-3/DssLitePsmMigrationPhase3.t.integration.sol

Findings

LOW

Fee boundary bug/footgun: tin == 100% allows selling gem for 0 Dai (no revert)

Locations:

`src/DssLitePsm.sol:283-297`

`src/DssLitePsm.sol:336-352`

Description:

`DssLitePsm.file("tin", data)` allows setting `tin` up to `1 * WAD` (100%). When `tin == WAD`, `_sellGem()` computes a fee equal to the full Dai output and silently returns 0 Dai while still transferring the user's `gem` to `pocket`.

Evidence:

Setting allows 100%:

```
if (what == "tin") {
 require(data == HALTED || data <= WAD, "DssLitePsm/tin-out-of-range");
 tin = data;
}
```

Sell path:

```
daiOutWad = gemAmt * to18ConversionFactor;
fee = daiOutWad * tin_ / WAD;
unchecked { daiOutWad -= fee; } // becomes 0 when tin_ == WAD

gem.transferFrom(msg.sender, pocket, gemAmt);
dai.transfer(usr, daiOutWad); // transfers 0
```

Concrete trace:

For a 6-decimal gem (e.g., USDC), `to18ConversionFactor = 10^(18-6) = 10^12`.

- `gemAmt = 100 * 10^6`
- `daiOutWad = gemAmt * 10^12 = 100 * 10^18`

- With `tin = 1 * 10^18`, `fee = daiOutWad`
- `daiOutWad -= fee => 0`

Transaction succeeds, user receives 0 Dai.

Impact:

A privileged actor (ward) can configure `tin` to a confiscatory value (100%) that causes **total loss** for any user calling `sellGem()` during that period.

Even if governance processes are expected to be timelocked externally, this is still an extremely sharp edge: the module already has a dedicated `HALTED` mechanism to disable flow; a 100% fee looks like a “working swap” but is effectively a silent confiscation.

Recommendation:

Consider forbidding the 100% value for `tin` (and potentially `tout`) unless explicitly required:

- Change the constraint to `data < WAD` (or a tighter max fee cap), while keeping `HALTED` as the explicit stop mechanism.
- Alternatively, revert in `_sellGem` when the computed `daiOutWad == 0 && gemAmt > 0` to avoid silent total-loss swaps.

LOW

Migration temporarily sets debt ceilings to uint256.max without validating post-migration ceilings invariants

Locations:

`src/deployment/DssLitePsmMigration.sol:124-126`

`src/deployment/DssLitePsmMigration.sol:160-167`

`src/deployment/DssLitePsmMigration.sol:181-185`

Description:

`DssLitePsmMigration.migrate()` temporarily sets both the global `Vat.Line` and the destination ilk `line` to `type(uint256).max`, then may call `dst.fill()` (which mints Dai via `vat.frob`). It then restores the previous ceilings.

There is **no validation** that the amount of Dai minted / the resulting destination ilk debt remains within the original ceilings after restoration.

This creates a validation gap where the migration can:

- Mint Dai while ceilings are unlimited,
- Restore lower ceilings,
- Leave the system **permanently above the configured risk limits**, causing later `frob`/`fill`-like operations to revert until governance intervenes.

Evidence:

Ceilings are forced to maximum:

```
dss.vat.file("Line", type(uint256).max);
dss.vat.file(dst.ilk, "line", type(uint256).max);
```

Then `fill()` can be called under those unlimited ceilings:

```
DssLitePsmLike(dst.psm).file("buf", mink);
if (DssLitePsmLike(dst.psm).rush() > 0) {
 DssLitePsmLike(dst.psm).fill();
}
```

Ceilings are then restored with no invariant check:

```
dss.vat.file("Line", currentGlobalLine);
dss.vat.file(dst.ilk, "line", dst.line);
```

Impact:

If the destination PSM is underfilled and/or if `mink` is large relative to the remaining headroom of the original ceilings, the migration can leave:

- The destination ilk above its `line` or
- The system above global `Line`

This can cause partial protocol DoS (further debt generation blocked) and violates governance-defined risk controls. Because `DSPause.exec` is permissionless after `eta`, third parties can choose the execution timing (within the allowed window), increasing the chance the state differs from what the config assumed.

Recommendation:

Add explicit pre/postcondition validation around the ceiling override:

- Compute the maximum Dai that could be minted during migration and require it fits within the original ceilings, OR
- After migration (before restoring ceilings), compute resulting debt and ensure it will not exceed the restored `Line`/`line`.
- Consider reverting if `mink == 0` to prevent accidental no-op migrations that still open a ceiling-bypass window.

LOW

Permissionless Phase 2 execution can be timed during collateral depeg to maximize value extraction (fee-free dst + immediate ceiling/buffer increase + fill)

Locations:

`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:122-145`

`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:129-134`

`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:139-145`

`src/deployment/phase-1/DssLitePsmMigrationPhase1.sol:99-113`

`src/deployment/DssLitePsmInit.sol:113-119`

`src/DssLitePsm.sol:411-476`

Description:

Phase 2 materially increases the *immediate* capacity and liquidity of the destination LitePSM in a way that can be **strategically timed** by external actors.

Key properties:

1. **Phase 2 is expected to be executed via the Maker `DSPauseProxy.exec` path, which is permissionless after `eta`.** This means *anyone* can choose the exact block/time the action is executed.
2. Phase 2 updates AutoLine for the destination ilk and immediately calls `exec()`:

```
autoLine.setIlk(res.dstIlk, cfg.dstMaxLine, cfg.dstGap, cfg.dstTtl);
autoLine.exec(res.dstIlk);
```

(`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:132-133`).

Because `setIlk()` resets AutoLine's `last/lastInc` timestamps to zero in the canonical Maker implementation, `exec()` can immediately raise the ilk's debt ceiling based on **current debt + gap**, effectively bypassing any prior TTL cadence.

3. Phase 2 then sets a **very large buffer** on the destination LitePSM and (conditionally) calls `fill()`:

```
DssLitePsmLike(res.dstPsm).file("buf", cfg.dstBuf);
if (DssLitePsmLike(res.dstPsm).rush() > 0) {
 DssLitePsmLike(res.dstPsm).fill();
}
```

(`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:139-145`).

`fill()` mints Dai into the LitePSM contract up to `rush()`, and `rush()` is derived from `gem.balanceOf(pocket)` and `buf` (among other ceilings):

```
uint256 tArt = gem.balanceOf(pocket) * to18ConversionFactor + buf;
wad = min( min(subcap(tArt, Art), subcap(line / RAY, Art)), subcap(vat.Line(), vat.debt()) /
RAY );
```

(`src/DssLitePsm.sol:463-475`).

4. **Destination LitePSM fees are not set by Phase 1 or Phase 2**, and default to `tin=0`/`tout=0` in `DssLitePsm`. This leaves the destination path effectively **fee-free** during/after Phase 2 unless another spell sets fees.

Exploit / Economic Attack Scenario:

If the collateral stablecoin (e.g., USDC) trades **below $1** (even temporarily), the PSM's fixed 1:1 pricing becomes an economically exploitable oracle-less "mispricing."

An attacker can:

1. Wait for a collateral depeg event (or short-lived liquidity dislocation).
2. **Execute Phase 2 immediately after `eta`** (permissionless), ensuring:
   - a large `buf` is configured,
   - `fill()` pre-mints substantial Dai liquidity,
   - AutoLine `exec()` may raise the destination ceiling immediately.
3. Immediately dump large amounts of underpriced collateral into the **fee-free** destination LitePSM (`sellGem`) to receive Dai at par.

This extracts value from the protocol (the system receives <$1 collateral while issuing $1 Dai), up to the newly unlocked ceilings/liquidity.

Impact:

High impact during adverse market conditions: Phase 2 can be used as a "liquidity unlock button" by *anyone* at the worst possible time, amplifying losses in a stablecoin depeg scenario.

The distinguishing issue here is not “PSMs are exposed to collateral depeg” (known design risk), but that:

- the spell is **permissionlessly executable**, and
- it **immediately increases** LitePSM capacity/liquidity and keeps it fee-free,

creating an attack window where adversaries can intentionally time execution to maximize extraction.

Recommendation:

- Treat Phase 2 execution as MEV-adversarial: avoid designs where the spell itself unlocks large immediate liquidity/ceiling *and* can be executed by any party at any time after `eta`.
- Consider sequencing/timelocking capacity increases (or making them conditional on safety checks) rather than doing `setIlk + exec + buf + fill` in one permissionlessly triggerable action.
- Ensure destination fees/spreads (or halting controls) are set appropriately before large buffers/ceilings are unlocked, to reduce profitable dumping during stress events.

LOW

Migration may unintentionally mint net new system debt and leave VAT Dai stuck in executor when `src.ink > src.art`

Location:

`src/deployment/DssLitePsmMigration.sol:137-192`

Description:

`DssLitePsmMigration.migrate()` allows `src.ink > src.art` (it only checks `src.ink >= src.art`).

In that case, it can migrate **collateral surplus** (`mink`) larger than the amount of debt removed from the source (`mart = min(src.art, mink)`).

The procedure:

1. `vat.grab(..., dink = -mink, dart = -mart)` reduces the source urn's `ink` by `mink` but reduces its `art` only by `mart`.
2. It sells the full `mink` worth of gem into the destination PSM, receiving `daiOutWad == mink`.
3. It converts that full `mink` Dai to VAT Dai via `daiJoin.join`.
4. It only calls `vat.heal(mart * RAY)`.

If `mink > mart`, after `heal` there will be leftover internal VAT Dai of `(mink - mart) * RAY` held by the executing address (`address(this)` in spell context, typically the PauseProxy), and the net global Maker debt can increase by the same amount (because the destination PSM minted `mink` Dai debt but only `mart` worth of "bad debt" was healed).

This behavior is inconsistent with the common expectation that a migration should not:

- leave the PauseProxy with a persistent VAT Dai balance, nor
- mint net new debt simply because the source PSM was overcollateralized.

Code Snippet:

```
uint256 mink = min(cfg.dstWant, subcap(src.ink, cfg.srcKeep));
uint256 mart = min(src.art, mink);

vat.grab(src.ilk, src.psm, address(this), address(this), -int256(mink), -int256(mart));
...
uint256 daiOutWad = dst.sellGemNoFee(address(this), srcGemAmt);
require(daiOutWad == mink, ...);

daiJoin.join(address(this), daiOutWad);
vat.heal(mart * RAY);
```

Impact:

If `src.ink > src.art` (e.g., due to donations, accounting quirks, or prior operations), the migration can:

- unexpectedly increase system debt, and
- strand surplus VAT Dai in the executor (PauseProxy), which may violate governance operational expectations and complicate accounting.

This is a governance/migration correctness risk that can cause the system to end in a state different from what the migration author intended.

Recommendation:

Either:

- enforce `src.ink == src.art` (or a tighter invariant) if that is required for correctness, **or**
- explicitly handle the `mink > mart` surplus case by directing/disposing of the extra `(mink - mart)` Dai in a well-defined way (e.g., move it to `vow`, or adjust `mink` so that migrated amount is debt-backed only).

LOW

Phase 2 intentionally reverts on low srcInk, enabling MEV/governance griefing DoS at execution time

Locations:

`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:98-121`

`src/deployment/DssLitePsmMigration.sol:146-152`

Description:

Phase 2 introduces an explicit post-migration invariant check that reverts if `srcPsm` ends below the desired keep amount:

```
(uint256 srcInk,) = dss.vat.urns(res.srcIlk, res.srcPsm);
require(srcInk >= cfg.srcKeep, "DssLitePsmMigrationPhase2/remaining-ink-too-low");
```

(`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:119-120`).

Because the migrated amount `mink` is computed from the *current* `src.ink` (`src/deployment/DssLitePsmMigration.sol:148`), an attacker can force this require to fail by reducing `srcPsm`'s `ink` below `cfg.srcKeep` immediately before governance execution.

Crucially, Maker spells are executed via `DSPauseProxy.exec` which is permissionless after `eta` and easily MEV-sandwiched. An attacker can front-run the execution attempt by trading against `srcPsm` to drop `src.ink < srcKeep` (potentially using flash liquidity and unwinding via deep stablecoin DEX pools). The spell then reverts, causing a **DoS** of the governance action.

Even if governance/keepers attempt to re-execute, the attacker can keep `src.ink` below the threshold at the relevant moments, repeatedly preventing execution.

Impact:

A denial-of-service vector against the Phase 2 migration spell at the exact time governance intends to execute it. This can delay risk migrations and force repeated governance interventions.

While the check was added to prevent a specific flash-loan manipulation, it also creates an explicit, user-influencable execution precondition that adversaries can exploit.

Recommendation:

- Avoid placing execution-critical `require()` conditions on values that are directly user-tradable/manipulable (`srcPsm` ink) when execution is permissionless and MEV-exposed.
- If such a check is necessary, add additional governance/operational mitigations (e.g., pre-conditioning steps that make manipulation costly *before* execution, or designing a migration that is robust to `src.ink` variability rather than reverting).

LOW

Phase-1/Phase-3 migrations never set LitePSM tin/tout despite init contract stating they must be set at higher level

Locations:

`src/deployment/DssLitePsmInit.sol:113-119`

`src/deployment/phase-1/DssLitePsmMigrationPhase1.sol:105-113`

`src/deployment/phase-3/DssLitePsmMigrationPhase3.sol:77-88`

Description:

`DssLitePsmInit.init()` explicitly states that `buf`, `tin`, and `tout` must be configured by higher-level migration scripts, but Phase 1 and Phase 3 only set `buf` on the destination LitePSM.

As a result, the destination LitePSM's fee parameters `tin`/`tout` remain at their constructor defaults (zero), meaning swaps are enabled with **0% fees** unless configured elsewhere.

This is an asset-flow risk: if the intended deployment requires non-zero fees (or intends to start in a halted state via `HALTED`), these migrations do not enforce it.

Evidence:

In `src/deployment/DssLitePsmInit.sol`:

```
// Notice: `buf`, `tin` and `tout` need to be set in the higher level migration scripts.
```

But in Phase 1 (`src/deployment/phase-1/DssLitePsmMigrationPhase1.sol`):

```
DssLitePsmLike(res.dstPsm).file("buf", cfg.dstBuf);
```

And in Phase 3 (`src/deployment/phase-3/DssLitePsmMigrationPhase3.sol`):

```
DssLitePsmLike(res.dstPsm).file("buf", cfg.dstBuf);
```

No `file("tin", ...)` or `file("tout", ...)` calls for the destination LitePSM in these phases.

Impact:

- If non-zero fees were intended during/after Phase 1 or Phase 3, the system can operate with **unexpectedly free swaps**, causing fee revenue loss and different economic behavior.
- If swaps were intended to be initially disabled (e.g., `tin/tout = HALTED`), this omission can unintentionally enable asset movement immediately after migration.

Recommendation:

Explicitly set `tin` and `tout` for the destination LitePSM in the migration phases (or remove/clarify the init comment if fees are guaranteed to be set elsewhere).

LOW

Reentrancy via token recipient hooks can drain PSM Dai buffer (no reentrancy guard, pocket self-transfer edge)

Locations:

`src/DssLitePsm.sol:310-352`

`src/DssLitePsm.sol:361-399`

Description:

`DssLitePsm` performs external token transfers in `sellGem()` / `buyGem()` with **no reentrancy protection**. While the contract keeps little internal state, its execution flow can still be exploited when interacting with hook-enabled tokens (ERC-777 / ERC-1363 / ERC-677-style callbacks) and/or a `pocket` contract that executes code upon receipt.

In particular, `_sellGem()` transfers `gem` to `pocket` **before** transferring Dai out. If the `gem.transferFrom()` triggers a callback on the `pocket` contract, `pocket` can re-enter the PSM mid-execution and call `sellGem()` itself. Because the PSM is expected to have approval to move `gem` on behalf of `pocket`, a re-entrant `sellGem()` where `msg.sender == pocket` will execute:

- `gem.transferFrom(pocket, pocket, gemAmt)` (a self-transfer that may be a net no-op for many tokens), then
- `dai.transfer(attacker, daiOutWad)`

This allows draining the PSM's pre-minted Dai buffer / fee Dai **without providing additional `gem`** (beyond what was provided by the original caller) as long as the outer call still has enough Dai to finish.

Vulnerable code:

`sellGem` / `_sellGem`:

```
function _sellGem(address usr, uint256 gemAmt, uint256 tin_) internal returns (uint256
daiOutWad) {
 ...
 gem.transferFrom(msg.sender, pocket, gemAmt);
 dai.transfer(usr, daiOutWad);
 emit SellGem(usr, gemAmt, fee);
}
```

`buyGem` also contains unguarded external calls that can be used for cross-function reentrancy if `dai`/`gem` are hook-enabled:

```
dai.transferFrom(msg.sender, address(this), daiInWad);
gem.transferFrom(pocket, usr, gemAmt);
```

Impact:

If a hook-enabled `gem` is configured and `pocket` is a contract capable of re-entering, an attacker controlling (or compromising) `pocket`'s callback code can drain the PSM's Dai balance (including pre-minted buffer and accumulated fees). This can cause:

- Loss of protocol-controlled Dai held by the PSM
- Forced depletion of liquidity, breaking swaps for users

Even if the intended deployment uses non-hook tokens (e.g., USDC/DAI), the contract does not enforce that assumption, so misconfiguration/upgrade of token implementations (or using this code with different tokens) can introduce a concrete exploit.

Recommendation:

Add explicit reentrancy protection around swap entrypoints (at least `sellGem*` and `buyGem*`) and/or harden execution flow against callback-based reentry (e.g., by restricting `pocket` to an EOA/no-code address, disallowing `msg.sender == pocket`, or using pull patterns / internal accounting).

LOW

cut() can underflow and revert, permanently breaking chug() fee forwarding in deficit states

Locations:

`src/DssLitePsm.sol:506-511`

`src/DssLitePsm.sol:443-453`

Description:

`DssLitePsm.cut()` computes:

`cash + gem.balanceOf(pocket) * to18ConversionFactor - art`

using checked arithmetic (Solidity ^0.8). If `art` is greater than `cash + gemValue`, this subtraction underflows and **reverts**. Since `chug()` calls `cut()`, `chug()` becomes permanently unusable whenever the PSM is in a deficit state.

A deficit state is reachable via realistic operational mistakes/misconfigurations, e.g.:

- `pocket` transfers gems away (or loses funds), reducing `gem.balanceOf(pocket)` while `art` remains.
- A non-standard/fee-on-transfer `gem` is configured (PSM pays out Dai based on `gemAmt`, but pocket receives less gem), allowing `cash + gemValue` to fall behind `art`.
- Any trusted-but-external component that can pull Dai from the PSM (the PSM pre-approves `daiJoin`) reduces `cash` without reducing `art`.

In these cases, the intended behavior is presumably “no fees to chug” (i.e., return 0), but the current implementation reverts instead.

Vulnerable Code:

```
function cut() public view returns (uint256 wad) {
 (, uint256 art) = vat.urns(ilk, address(this));
 uint256 cash = dai.balanceOf(address(this));

 wad = _min(cash, cash + gem.balanceOf(pocket) * to18ConversionFactor - art);
}
```

Impact:

- **DoS of fee forwarding:** `chug()` will revert forever while `cash + gemValue < art`, so accumulated fees (or donated Dai intended to be forwarded) cannot be moved to `vow`.
- **Integration fragility:** `cut()` is a public view function; callers (UIs/keepers/spells) can be forced into unexpected reverts.

Recommendation:

Make `cut()` non-reverting for deficit states by saturating the subtraction to zero (e.g., `_subcap(cash + gemValue, art)`) and then taking `min(cash, ...)`, or equivalent safe math. This preserves the intended "no fees available" behavior instead of reverting.

LOW

cut()/chug() ignore Vat rate; if stability fee ever becomes non-zero, chug can mis-account and worsen deficit

Locations:

`src/DssLitePsm.sol:463-496`

`src/DssLitePsm.sol:506-511`

`src/DssLitePsm.sol:443-452`

Description:

The contract assumes (and partially enforces) that the Vat `rate` for the `ilk` is always exactly `RAY` (i.e., stability fee always zero). `rush()` and `gush()` enforce this with `require(rate == RAY)`.

However, `cut()` (used by permissionless `chug()`) does **not** check `rate` and subtracts the normalized debt `art` directly from `cash + gemValue`:

`cash + gemValue - art`

In Maker's Vat, the true debt in `rad` is `art * rate`. If `rate > RAY` (non-zero stability fee), the PSM's *economic* surplus is smaller than `cash + gemValue - art`.

As a result, after a rate increase, `chug()` can still be called permissionlessly and can move DAI to `vow` based on an outdated 1:1 debt assumption, even if the PSM is actually in deficit when accounting for `rate`.

Code Snippet:

```
function rush() public view returns (uint256 wad) {
 (uint256 Art, uint256 rate,, uint256 line,) = vat.ilks(ilk);
 require(rate == RAY, "DssLitePsm/rate-not-RAY");
 ...
}

function cut() public view returns (uint256 wad) {
 (, uint256 art) = vat.urns(ilk, address(this));
 uint256 cash = dai.balanceOf(address(this));
 wad = _min(cash, cash + gem.balanceOf(pocket) * to18ConversionFactor - art);
}
```

Impact:

If governance (or an upstream system change) ever causes `rate != RAY`:

- `fill()`/`trim()` become unusable (intended safeguard), but
- `chug()` remains callable and may send DAI out as “fees” even though the PSM’s position is effectively accruing stability-fee debt.

This can worsen an already-deficit position and complicate accounting/recovery.

Recommendation:

Either:

- enforce `rate == RAY` inside `cut()`/`chug()` as well, or
- account for `rate` when computing the debt term (convert `art` to wad-equivalent using `rate`).

LOW

Migration temporarily sets Vat debt ceilings to uint256.max; callback/reentrancy can borrow beyond global ceiling

Location:

`src/deployment/DssLitePsmMigration.sol:159-184`

Description:

`DssLitePsmMigration.migrate()` temporarily sets Maker's **global debt ceiling** (`Vat.Line`) and the destination ilk's **per-ilk ceiling** (`vat.ilks[dst].line`) to `type(uint256).max` to accommodate migration:

```
dss.vat.file("Line", type(uint256).max);
dss.vat.file(dst.ilk, "line", type(uint256).max);
```

While these ceilings are lifted, the function performs multiple **external calls** (e.g., `GemJoin.exit`, `dstPsm.fill`, `dstPsm.sellGemNoFee`, token `approve/transfer/transferFrom`). If any of these calls can trigger attacker-controlled code execution (e.g., via a malicious/upgradeable collateral token, ERC777-style hooks, non-standard token callbacks, or a compromised join adapter), that callback can execute **`Vat.frob()`-based borrowing** while `Vat.Line` is temporarily unlimited.

Because the migration later resets ceilings back down, the attacker's newly created debt can remain **above the intended global ceiling** (Maker allows ceilings to be set below current debt; it only blocks *new* debt).

Impact:

- **Economic/risk-limit bypass:** attacker can mint Dai beyond the configured global ceiling during the lifted-ceiling window.
- After ceilings are restored, the system may be left in an over-ceiling state, impairing risk controls and potentially causing system-wide debt issuance DoS until governance intervenes.

Recommendation:

Avoid setting `Vat.Line` to `uint256.max` during sequences that include external calls. If temporary headroom is required, bound it tightly (e.g., raise by the minimal delta needed) and/or structure the migration to minimize/avoid untrusted external calls while ceilings are lifted.

LOW

Integration test 'testTrim_Revert_WhenRateIsInvalid' calls fill() instead of trim(), leaving trim path untested

Location:

`src/DssLitePsm.t.integration.sol:543-548`

Description:

The test named `testTrim_Revert_WhenRateIsInvalid()` intends to validate that `trim()` reverts when `vat.ilks(ilk).rate != RAY`, but it actually calls `litePsm.fill()`:

```
function testTrim_Revert_WhenRateIsInvalid() public {
 dss.vat.fold(ilk, address(111), 1);

 vm.expectRevert("DssLitePsm/rate-not-RAY");
 litePsm.fill();
}
```

Impact:

- The `trim()` rate-invalid revert behavior is **not tested**, reducing confidence in numerical assumptions around the `rate == RAY` invariant.
- Potential arithmetic/precision issues in `gush()`/`trim()` could slip through because the intended negative test does not execute the target function.

Recommendation:

Call `litePsm.trim()` in this test (and keep/adjust `fill()` coverage in the dedicated fill test).

LOW

DssLitePsmInit.init gas usage depends on unbounded token name/symbol strings (risk of OOG / unexecutable spell)

Location:

`src/deployment/DssLitePsmInit.sol:126-138`

Description:

`DssLitePsmInit.init()` fetches `GemLike(cfg.gem).name()` and `GemLike(cfg.gem).symbol()` and passes them into `IlkRegistry.put()`. Both calls are external and can return arbitrarily large strings (or be implemented with expensive logic). The subsequent `put()` will also need to copy/store those strings, making gas cost roughly proportional to returned string length.

Since `init()` is intended to be executed on-chain as part of a governance spell (via `DSPauseProxy.exec`/`delegatecall`), this creates a parameter-dependent gas bound: a misconfigured or nonstandard `gem` can cause the init transaction/spell execution to exceed the block gas limit and revert, preventing the PSM from being initialized and potentially requiring re-deployment/re-scheduling.

Evidence:

```
// src/deployment/DssLitePsmInit.sol
IlkRegistryLike reg = IlkRegistryLike(dss.chainlog.getAddress("ILK_REGISTRY"));
reg.put(
 cfg.ilk,
 address(0),
 cfg.gem,
 GemLike(cfg.gem).decimals(),
 REG_CLASS_JOINLESS,
 cfg.pip,
 address(0),
 GemLike(cfg.gem).name(),
 GemLike(cfg.gem).symbol()
);
```

Impact:

- Governance initialization can become **unexecutable** (out-of-gas) depending on `gem` implementation and metadata string lengths.
- Delays or blocks migrations/deployments; requires operational recovery (new spell, different config, or omitting registry write).

Recommendation:

- Avoid on-chain fetching/storing of unbounded strings during critical initialization.
- Prefer bounded/constant metadata supplied in config (pre-validated length caps), or store only fixed-size identifiers (e.g., bytes32) on-chain.
- Consider guarding against unexpectedly large return data (length caps) before calling `IlkRegistry.put()`.

LOW

Init sets virtual `ink` based on constant RAY without enforcing actual `spot`/`par`, risking overflow/DoS when `par` or `spot` != 1.0

Locations:

`src/deployment/DssLitePsmInit.sol:104-112`

`src/DssLitePsm.sol:48-54`

`src/DssLitePsm.sol:463-476`

Description:

`DssLitePsmInit.init()` seeds the Vat urn with a huge "virtual collateral" amount `vink = type(uint256).max / RAY` under the assumption that the ilk spot price is exactly `RAY` (1.0) and that `spotter.par == RAY`.

However, `init()` **does not verify** that `spotter.par` is `RAY` (nor that the resulting Vat `spot` is `RAY`) before sizing `vink`. If `par` is ever set above 1.0 (or if other configuration makes `vat.ilks[ilk].spot > RAY`), then downstream Vat math that multiplies `ink * spot` can overflow because `ink` was chosen using `RAY` rather than the actual `spot`.

This breaks core numerical invariants and can cause `fill()`/`trim()` to revert (permanent until governance intervention), effectively DoSing the Lite PSM.

Evidence:

`init()` uses a constant denominator:

```
// Set `ink` to the largest value that will not cause an overflow for `ink * spot`.
// Notice: `litePsm` assumes that:
//   a. `spotter.par == RAY`
//   b. `vat.ilks[ilk].spot == RAY`
int256 vink = int256(type(uint256).max / RAY);
dss.vat.slip(cfg.ilk, inst.litePsm, vink);
dss.vat.grab(cfg.ilk, inst.litePsm, inst.litePsm, address(0), vink, 0);
```

But the contract itself documents the same assumptions without enforcing them:

```
// 3. The `spot` price for gem is always 1 (`10**27`).
// 4. The `spotter.par` (Dai parity) is always 1 (`10**27`).
```

`fill()` relies on the urn being usable for minting:

```
vat.frob(ilk, address(this), address(0), address(this), 0, _int256(wad));
```

If `spot` is higher than assumed, Vat's internal collateralization math (which multiplies `ink` by `spot`) can overflow and revert.

Impact:

- **High impact DoS:** `fill()`/`trim()` can become unusable, preventing the module from maintaining required Dai liquidity.
- The failure can be triggered by realistic governance/system state changes (e.g., `spotter.par` adjustments) rather than only by malicious callers.

Recommendation:

During initialization, enforce the numeric assumptions used to size `vink` (e.g., assert `spotter.par == RAY` and `vat.ilks[ilk].spot == RAY` after `poke()`), or compute `vink` from the **actual** `spot` (`type(uint256).max / spot`) with an additional safety margin.

LOW

Migration can revert due to conversion-factor dust because it never validates `src.ink`/computed `mink` divisibility by `to18ConversionFactor`

Locations:

`src/deployment/DssLitePsmMigration.sol:129-158`

`src/deployment/DssLitePsmMigration.sol:169-173`

Description:

`DssLitePsmMigration.migrate()` converts the wad-denominated amount to migrate (`mink`) into a gem-denominated amount via integer division:

```
uint256 srcGemAmt = mink / to18ConversionFactor;
...
uint256 daiOutWad = DssLitePsmLike(dst.psm).sellGemNoFee(address(this), srcGemAmt);
require(daiOutWad == mink, "DssLitePsmMigration/invalid-dai-amount");
```

This implicitly assumes `mink % to18ConversionFactor == 0`.

The code only checks divisibility for config inputs (`dstWant`, `srcKeep`), but **does not** check the on-chain `src.ink` (or the computed `mink`) is divisible by `to18ConversionFactor`.

Even if both `dstWant` and `srcKeep` are multiples of the conversion factor, `mink = min(dstWant, src.ink - srcKeep)` can still be non-divisible if `src.ink` contains dust (e.g., due to prior `vat.slip/grab` operations, nonstandard joins, or manual Vat adjustments). In that case, `srcGemAmt` truncates, `sellGemNoFee` returns a smaller amount, and the `require(daiOutWad == mink)` reverts.

This issue is **broader** than the special-case `dstWant == type(uint256).max`.

Impact:

- **Migration DoS / fragility:** a small amount of misaligned `src.ink` dust can brick the migration spell, requiring rescheduling with a different approach.

Recommendation:
Validate the actual migrated amount is conversion-safe, e.g. require `src.ink % to18ConversionFactor == 0` (and/or `mink % to18ConversionFactor == 0`) before performing `mink / to18ConversionFactor`, or compute `mink` as an aligned amount (rounding down with explicit behavior) and use that aligned amount consistently.

LOW

Permanent DoS risk: `fill()`/`trim()`/`rush()`/`gush()` hard-require `vat.ilks(ilk).rate == RAY`

Locations:

`src/DssLitePsm.sol:463-476`

`src/DssLitePsm.sol:482-496`

`src/DssLitePsm.sol:411-420`

`src/DssLitePsm.sol:428-437`

Description:

`rush()` and `gush()` enforce `rate == RAY`:

```
(uint256 Art, uint256 rate,, uint256 line,) = vat.ilks(ilk);
require(rate == RAY, "DssLitePsm/rate-not-RAY");
```

This makes `fill()` and `trim()` unusable if `rate` is ever changed from `1e27`.

In Maker, `rate` is **monotonic increasing** once stability fees are enabled/dripped (or if `vat.fold()` is used). If `rate` becomes `> RAY` even once, it generally cannot be restored to exactly `RAY`.

Impact:

If the `ilk` ever accrues any stability fee (even accidentally via governance/config), then:

- `rush()`/`gush()` revert permanently,
- `fill()`/`trim()` revert permanently,
- the PSM can no longer be refilled with Dai liquidity, so `sellGem()` eventually becomes unusable once the existing Dai buffer is drained.

This is a long-lived, potentially permanent operational DoS of a core function (maintaining swap liquidity).

Recommendation:

If the intended invariant is truly “rate must always be RAY”, enforce it at the system level (prevent setting duty != RAY, prevent dripping, etc.) and/or make the contract robust to `rate != RAY` by incorporating `rate` correctly into its accounting rather than hard-reverting.

At minimum, add explicit deployment-time assertions and monitoring for any possible `rate` deviation.

LOW

Phase 2 migration can be economically griefed into migrating ~0 funds (srcInkHsrcKeep) despite intended flash-loan deterrence

Locations:

`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:57-146`

`src/deployment/DssLitePsmMigration.sol:105-192`

`src/deployment/phase-1/DssLitePsmMigrationPhase1.sol:61-113`

`src/deployment/DssLitePsmInit.sol:113-119`

Description:

`DssLitePsmMigrationPhase2.migrate()` is intended to perform a “major migration of funds” from `srcPsm` to `dstPsm` (via `DssLitePsmMigration.migrate()`), while keeping `cfg.srcKeep` collateral in the source PSM.

However, the amount migrated is computed directly from the *current* `srcPsm` urn collateral (`src.ink`) at execution time:

- `mink = min(dstWant, max(src.ink - srcKeep, 0))` (`DssLitePsmMigration.migrate`, `src/deployment/DssLitePsmMigration.sol:148`).
- Phase 2 uses `dstWant = type(uint256).max`, so `mink` becomes `max(src.ink - srcKeep, 0)`.

This makes the migration economically manipulable: an attacker can (potentially via flash-loaned DAI and deep DEX liquidity) reduce the source PSM’s `ink` to exactly `srcKeep` immediately before (or inside the same tx as) calling the permissionless `DSPauseProxy.exec` for the spell. The migration then deterministically moves **0** collateral while still applying Phase 2 configuration changes.

The code attempts to deter a flash-loan scenario by requiring `cfg.srcTin > 0` and `cfg.srcTout > 0`:

```
require(cfg.srcTin > 0, ".../src-tin-is-zero");
require(cfg.srcTout > 0, ".../src-tout-is-zero");
```

(`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:70,85`).

But this does **not** reliably enforce the intended deterrence:

1. **The attacker can drain `srcPsm` before the spell applies the new fees.** Phase2 sets `srcTin/srcTout` only at the end (`file("tin"/"tout")`), so the "drain" leg can occur with whatever fees are active pre-spell (often 0 in real deployments).

2. **The deterrence assumes the attacker must unwind via `srcPsm` post-spell (paying `srcTin`).** The attacker can instead unwind via other venues (e.g., DEXes or the new LitePSM if it has low/zero fees) to repay the flashloan, avoiding the fee the comments rely on.

3. **No minimum fee is enforced.** Requiring `> 0` allows governance to set a trivially small fee that does not meaningfully disincentivize the attack.

Additionally, `DssLitePsmInit` explicitly whitelists the PauseProxy for `sellGemNoFee` on the LitePSM (`kiss(address(this))` under delegatecall), and neither Phase1 nor Phase2 set nonzero `tin/tout` on the destination LitePSM (they only set `buf`). This increases the likelihood that an attacker can unwind through `dstPsm` cheaply during the migration window.

Impact:

High-likelihood governance/economic griefing: the Phase 2 spell can be executed successfully while migrating near-zero collateral, leaving the new LitePSM under-funded relative to governance intent. This can:

- Delay or prevent the intended risk migration from `srcPsm` to `dstPsm`.
- Force governance into additional emergency actions (re-scheduling spells, adjusting fees, replenishing) while the system remains in an unintended exposure state.
- Create a time window where `dstPsm` parameters are updated (e.g., higher ceilings/buffer) without receiving the intended migrated collateral, changing system economics in a way governance did not intend.

Recommendation:

- Don't rely on "`fee > 0`" as the sole deterrent for a state-dependent migration amount. Enforce migration progress more directly (e.g., require a minimum `mink` migrated, or compute the migrated amount from a reference snapshot / invariant not manipulable at exec time).
- If deterrence by fees is the goal, enforce a **minimum meaningful** fee (and/or apply it before the attacker can perform the drain leg), and consider that attackers can unwind via alternative venues.

- Consider splitting parameter changes from the migration of funds, so a “0-migration” execution cannot still apply the rest of Phase 2 changes.

LOW

fill()/trim() use global ilk.Art and assume only one urn exists; violates accounting if multiple urns created

Location:

`src/DssLitePsm.sol:463-496`

Description:

`rush()` and `gush()` compute mint/burn amounts using `vat.ilks(ilk).Art` (the *global* normalized debt for the ilk) and explicitly assume `urn.art == ilk.Art` ("no other urns for the same `ilk`"). However, `fill()` / `trim()` apply the resulting `wad` to **this contract's** urn via `vat.frob(ilk, address(this), ...)`.

If the assumption is ever violated (e.g., governance/automation creates another urn for the same ilk, or a future migration leaves residual debt elsewhere), the module's rebalancing logic becomes incorrect:

- `fill()` can under-mint (or mint nothing) even when this LitePSM urn needs Dai liquidity (DoS of `sellGem` once Dai balance is drained).
- `trim()` can attempt to burn more than this urn's actual `art`, causing `vat.frob` to revert and making trimming (and thus debt-ceiling recovery) fail.

This is a business-logic fragility: the contract relies on an external invariant but does not enforce it, yet uses it for core accounting.

Evidence:

`rush()`:

```
(uint256 Art, uint256 rate,, uint256 line,) = vat.ilks(ilk);
...
// To avoid two extra SLOADs it assumes urn.art == ilk.Art.
_subcap(tArt, Art)
```

`gush()`:

```
(uint256 Art, uint256 rate,, uint256 line,) = vat.ilks(ilk);
...
// To avoid two extra SLOADs it assumes urn.art == ilk.Art.
_subcap(Art, tArt)
```

But `fill()`/`trim()` frob **this** urn:

```
vat.frob(ilk, address(this), address(0), address(this), 0, _int256(wad));
...
vat.frob(ilk, address(this), address(0), address(this), 0, -_int256(wad));
```

Impact:

High: Incorrect rebalancing can lead to a persistent inability to `fill()`/`trim()`, which in turn can cause loss of core functionality (swaps halting due to insufficient Dai liquidity) and/or inability to reduce debt after buy pressure.

Recommendation:

Enforce the “single-urn” invariant on-chain (or stop relying on it): use `vat.urns(ilk, address(this)).art` for accounting, or add explicit checks that global `Art` equals this urn’s `art` before proceeding.

INFO

Balance-based bookkeeping (`rush/gush/cut`) is manipulable via direct token transfers, enabling liquidity griefing and unexpected fee realization

Locations:

`src/DssLitePsm.sol:463-475`

`src/DssLitePsm.sol:482-495`

`src/DssLitePsm.sol:506-511`

`src/DssLitePsm.t.integration.sol:809-889`

Description:

`DssLitePsm` derives mint/burn capacity (`rush/gush`) and fee realizability (`cut`) directly from external token balances:

- `gem.balanceOf(pocket)`
- `dai.balanceOf(address(this))`

Any address can change these balances via **direct transfers** (“donations”), without going through swap functions.

While some donation behavior may be acceptable/expected, it creates an economic attack surface where an attacker can manipulate bookkeeping outcomes and reduce/shift available liquidity at specific times.

Evidence:

Bookkeeping relies on raw balances:

```
// src/DssLitePsm.sol
uint256 tArt = gem.balanceOf(pocket) * to18ConversionFactor + buf; // rush/gush
...
uint256 cash = dai.balanceOf(address(this));
wad = _min(cash, cash + gem.balanceOf(pocket) * to18ConversionFactor - art); // cut
```

The integration test explicitly reproduces donation-driven behavior changes:

- `testGemDonation_Reproduce()` shows that donating `gem` to `pocket` changes `chug()`/`fill()` behavior.
- `testDaiDonation_Reproduce()` shows that donating Dai affects `chug()` and interacts with `trim()`/`fill()` constraints.

Impact:

- **Liquidity griefing / MEV:** attackers can time donations + permissionless `chug/trim/fill` to change the available Dai liquidity before victims' swaps, increasing revert probability and wasting gas.
- **Unexpected protocol accounting:** unsolicited balance changes are treated as fee surplus by `cut()`, potentially causing surprising `chug()` behavior (e.g., draining available Dai liquidity into `vow`).

Although these attacks generally require donating real assets (so they are not usually directly profitable), they can still be used for strategic MEV griefing or to disrupt integrators relying on predictable liquidity.

Recommendation:

- Treat all balance-derived values as attacker-influencable: document this clearly for integrators.
- Consider adding explicit "sync"/accounting mechanisms or donation-handling rules if predictable behavior is required.
- Encourage callers doing large swaps to bundle `fill()` (as README already suggests) and/or add wrapper-level slippage/revert management.

INFO

Phase2 does not validate srcKeep against intended migration amount; srcKeep==0 allows full migration and srcKeep==srcInk allows silent no-op migration

Locations:

`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:87-96`

`src/deployment/DssLitePsmMigration.sol:146-151`

`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:119-121`

Description:

Phase2's behavior depends heavily on `cfg.srcKeep`, but the only postcondition enforced is:

```
require(srcInk >= cfg.srcKeep, ...);
```

This does **not** ensure that Phase2 actually migrates a "major" amount of funds, nor that it leaves a nonzero amount behind.

In the underlying migration library:

- The amount migrated is `mink = min(dstWant, subcap(srcInk, srcKeep))`.
- If `srcKeep == 0`, then `mink` becomes essentially `srcInk` (full migration).
- If `srcKeep == srcInk`, then `subcap(srcInk, srcKeep) == 0` and `mink == 0` (no funds migrate), yet Phase2's `srcInk >= srcKeep` check still passes.

As a result, a configuration mistake (or an unexpected pre-state where `srcInk` has fallen to `srcKeep`) can cause Phase2 to succeed while migrating **nothing**, or to migrate **everything**, which may contradict the intended "major migration but keep liquidity" plan.

Evidence:

Phase2 forwards srcKeep and only checks `>=`:

```
MigrationResult memory res = DssLitePsmMigration.migrate(
 dss,
 MigrationConfig({ srcPsmKey: cfg.srcPsmKey, dstPsmKey: cfg.dstPsmKey, srcKeep: cfg.srcKeep,
dstWant: type(uint256).max })
);
...
(uint256 srcInk,) = dss.vat.urns(res.srcIlk, res.srcPsm);
require(srcInk >= cfg.srcKeep, "DssLitePsmMigrationPhase2/remaining-ink-too-low");
```

(`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:87-96,119-121`)

Migration amount is derived from srcKeep:

```
uint256 mink = _min(cfg.dstWant, _subcap(src.ink, cfg.srcKeep));
```

(`src/deployment/DssLitePsmMigration.sol:148`)

Impact:

Informational (misconfiguration risk):

- Phase2 can unintentionally perform a full migration (if `srcKeep == 0`).
- Phase2 can unintentionally do nothing (if `srcKeep == srcInk`), while still changing AutoLine and fee parameters.

Recommendation:

Add explicit postconditions matching the intended operation, e.g. require `res.sap > 0` (or above a minimum), and/or require `cfg.srcKeep > 0` if keeping liquidity in the source PSM is a hard requirement.

INFO

Phase2 does not validate that MCD_IAM_AUTO_LINE chainlog lookup returns a contract address (nonzero/code), risking misconfiguration DoS

Location:

`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:122-127`

Description:

Phase2 resolves the AutoLine address via `dss.chainlog.getAddress("MCD_IAM_AUTO_LINE")` and immediately casts it to `AutoLineLike`.

There is no validation that the returned address is:

- nonzero, and
- contains contract code.

If the chainlog entry is missing/repointed (or the `dss` instance is constructed incorrectly in a spell environment), the subsequent `setIlk`/`exec` calls will revert with opaque low-level errors, making the governance action unexecutable.

Evidence:

```
AutoLineLike autoLine = AutoLineLike(dss.chainlog.getAddress("MCD_IAM_AUTO_LINE"));

autoLine.setIlk(res.srcIlk, cfg.srcMaxLine, cfg.srcGap, cfg.srcTtl);
autoLine.exec(res.srcIlk);
```

(`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:123-127`)

Impact:

Informational:
primarily affects spell robustness (misconfiguration/registry risk). A bad chainlog entry can halt migration execution until governance fixes and re-schedules.

Recommendation:
Validate `autoLine` is nonzero and `autoLine.code.length > 0` (or equivalent) before calling it, and revert with a clear error message if invalid.

INFO

Phase2 integration test assumes AutoLine.exec always updates `last`/`lastInc`, but exec can be a no-op leaving them at zero

Locations:

`src/deployment/phase-2/DssLitePsmMigrationPhase2.t.integration.sol:314-321`

`src/deployment/phase-2/DssLitePsmMigrationPhase2.t.integration.sol:339-347`

Description:

`DssLitePsmMigrationPhase2.t.integration.sol` asserts that `DssAutoLine.exec()` always performs an update (and therefore sets `last` to `block.number` and `lastInc` to `block.timestamp`).

However, Maker's `DssAutoLine.exec()` **returns early without modifying storage** when the computed `lineNew` equals the current Vat `line`, or when the TTL gate prevents an increase. This is especially important because `setIlk()` resets `(last, lastInc)` to `(0,0)`; if the subsequent `exec()` is a no-op, those values remain `0`.

So the test encodes an incorrect semantic assumption about `exec()` and can fail (or miss edge behavior) under realistic chain states where `lineNew == line`.

Evidence:

Test asserts `last == block.number` for `SRC_ILK`:

```
(uint256 maxLine, uint256 gap, uint48 ttl, uint256 last,) = autoLine.ilks(SRC_ILK);
...
assertEq(last, block.number, "after: AutoLine invalid last");
```

Test asserts `lastInc == block.timestamp` for `DST_ILK`:

```
(uint256 maxLine, uint256 gap, uint48 ttl, uint256 last, uint256 lastInc) = autoLine.ilks(DST_ILK);
...
assertEq(lastInc, block.timestamp, "after: AutoLine invalid lastInc");
```

But `DssAutoLine.exec()` may early-return without updating `last`/`lastInc` when `lineNew == line` (or TTL blocks increase).

Impact:

- The test can be brittle against mainnet-fork state changes (e.g., if the Vat `line` already equals the computed `lineNew`).
- It may give a false sense of correctness by assuming `exec()` always mutates state, whereas in production the same call sequence can leave AutoLine time-stamps/blocks at zero after `setIlk()`.

Recommendation:

Relax the assertions to match `DssAutoLine.exec()` semantics (e.g., allow `last-`/`lastInc` to remain `0` when `exec()` is a no-op), or explicitly set up preconditions guaranteeing that `exec()` must change the Vat line in these tests.

INFO

Phase2 integration test assumes LitePSM Dai balance equals `buf` exactly after migration; LitePSM logic does not guarantee equality

Location:

`src/deployment/phase-2/DssLitePsmMigrationPhase2.t.integration.sol:323-337`

Description:

The Phase2 integration test asserts the LitePSM's ERC-20 Dai balance equals the configured `dstBuf` exactly after Phase2.

Semantically, this is not guaranteed by `DssLitePsm`:

- `buf` is an internal parameter used in `rush()` as part of a target `tArt = gemBalance(pocket)*to18 + buf`.
- `fill()` mints `wad = rush()` and exits that amount of Dai, but **it does not attempt to set `dai.balanceOf(litePsm) == buf`**.
- The LitePSM Dai balance can be influenced by prior swaps, donations, or other flows; Phase2 also does not call `trim()` to burn excess.

So the assertion encodes a stronger invariant than the implementation provides.

Evidence:

Test expects exact equality:

```
assertEq(dss.dai.balanceOf(address(dstPsm)), mig2Cfg.dstBuf, "after: invalid dst psm dai balance");
```

But `DssLitePsm.fill()` mints based on `rush()` (which uses `Art`, `line`, global ceilings, pocket gem balance, and `buf`), not based on the current ERC-20 Dai balance.

Impact:

The test can become flaky across mainnet forks or future system states (e.g., if the LitePSM contract has a non-zero unexpected Dai balance before Phase2). It may also hide scenarios where Phase2 correctly sets `buf` but the post-state Dai balance differs for legitimate reasons.

Recommendation:
Use a weaker assertion aligned with contract semantics (e.g., `>= dstBuf` if the intent is “at least buf liquidity”, or assert on `rush()`/`gush()`/`Art`-based invariants instead of ERC-20 Dai balance), or explicitly normalize the Dai balance (via `trim`) if exact equality is truly required.

INFO

Phase2 integration test assumes source PSM `art` decreases by the moved `ink`; migration logic caps debt reduction at `min(src.art, movedInk)`

Location:

`src/deployment/phase-2/DssLitePsmMigrationPhase2.t.integration.sol:300-311`

Description:

In `_checkMigrationPhase2()`, the test asserts the source PSM's normalized debt (`art`) decreases by `expectedMoveWad` (computed from `ink` and `srcKeep`).

But the underlying migration implementation (`DssLitePsmMigration.migrate`) explicitly computes:

- `mink = min(dstWant, subcap(src.ink, srcKeep))`
- `mart = min(src.art, mink)`

and performs `vat.grab(..., -mink, -mart)`, meaning **the debt reduction is `mart`, not necessarily `mink`**.

So the test encodes an extra assumption that `src.art >= mink` (effectively `src.art` tracks `src.ink` closely enough), which is not guaranteed by the migration logic itself.

Evidence:

Test assumption:

```
assertEq(srcArt, psrcArt - expectedMoveWad, "after: src art is not decreased by the moved amount");
```

But migration uses capped debt reduction:

```
uint256 mink = _min(cfg.dstWant, _subcap(src.ink, cfg.srcKeep));
uint256 mart = _min(src.art, mink);
dss.vat.grab(src.ilk, src.psm, address(this), address(this), -_int256(mink), -_int256(mart));
```

Impact:

- The test may underflow/revert or fail unexpectedly if `src.art < expectedMoveWad` (even though the migration code would still behave consistently by capping `mart`).
- The test may miss edge cases where the source PSM's `ink` and `art` diverge (whether due to rounding, governance interventions, or non-standard PSM implementations).

Recommendation:

Adjust the assertion to match the migration semantics (i.e., `srcArt == psrcArt - min(psrcArt, expectedMoveWad)`), or explicitly constrain the test setup to guarantee `psrcArt >= expectedMoveWad` and document that assumption.

INFO

Phase3 integration test comments about which phase's max line is reached appear inconsistent with configured values

Location:

```
src/deployment/phase-3/DssLitePsmMigrationPhase3.t.integra-
tion.sol:214-238
```

Description:

The Phase 3 integration tests include comments claiming the destination `line` reaches "the max value defined by phase 1/phase 2":

- `testMigrationPhase3WhenSrcIsEmpty`: "max value defined by phase 1"
- `testMigrationPhase3WhenDstLineIsMax`: "max value defined by phase 2"

However, in `setUp()` the configured max lines differ by phase (`mig1Cfg.dstMaxLine`, `mig2Cfg.dstMaxLine`, `mig3Cfg.dstMaxLine`), and Phase 3 explicitly calls `autoLine.setIlk(..., mig3Cfg.dstMaxLine, ...)`.

These comments are therefore misleading about what the test is actually asserting (the test doesn't directly assert the Vat `line` value at all; it mostly asserts global line deltas and AutoLine params).

Impact:

- **False confidence / maintenance hazard:** future readers may misunderstand what these tests guarantee.

Recommendation:

Update comments (or add explicit assertions on `vat.ilks(DST_ILK).line`) so the test intent matches what is being checked.

INFO

Swap entrypoints accept `gemAmt = 0`, causing no-op transfers and misleading events (spam/griefing vector)

Location:

`src/DssLitePsm.sol:310-399`

Description:

`sellGem*` / `buyGem*` accept `gemAmt` without validating it is non-zero.

Calling with `gemAmt == 0` results in:

- fee computed as 0
- transfers of 0 tokens (usually succeed on ERC-20)
- an event emission (`SellGem`/`BuyGem`) that may be interpreted by off-chain systems as a meaningful trade

This is an input validation gap on user-controlled amounts.

Evidence:

No `require(gemAmt > 0)` exists in any swap entrypoint.

Impact:

Primarily off-chain impact: event spam, misleading analytics/accounting, and potential griefing (wasting indexer resources).

Recommendation:

Require `gemAmt > 0` in swap entrypoints (or explicitly document that 0-amount swaps are allowed and should be filtered off-chain).

INFO

Test helper _changeIlkLine() can underflow when decreasing line by more than current line (negative dline), causing unintended reverts

Location:

`src/DssLitePsm.t.integration.sol:1054-1071`

Description:

`_changeIlkLine(bytes32 ilk_, int256 dline, bool global)` computes:

```
uint256 lineNew = dline > 0 ? line + uint256(dline) : line - uint256(-dline);
```

If `dline < 0` and `uint256(-dline) > line`, the subtraction underflows and reverts.

Impact:

- Makes the helper unsafe for fuzzing or future test extensions that attempt to reduce `line` aggressively.

Recommendation:

Use a capped subtraction (or explicit `require(uint256(-dline) <= line, ...)`) to avoid accidental underflow.

INFO

`ARITHMETIC_ERROR` strings do not actually trigger Solidity panic; `_int256` revert behavior differs from comment/intent

Locations:

`src/deployment/DssLitePsmMigration.sol:80-88`

`src/DssLitePsm.sol:95-101`

`src/DssLitePsm.sol:197-201`

Description:

Both `DssLitePsmMigration` and `DssLitePsm` attempt to “explicitly revert with an arithmetic error” by using:

```
string internal constant ARITHMETIC_ERROR = string(abi.encodeWithSignature("Panic(uint256)",
0x11));
```

and then:

```
require((y = int256(x)) >= 0, ARITHMETIC_ERROR);
```

This does **not** produce a Solidity `Panic(0x11)` revert. `require` always reverts with `Error(string)` encoding, so callers and tooling will observe a standard revert string containing binary data, not a panic.

Impact:

- Off-chain tooling or tests that expect a true panic (or a readable revert reason) may mis-handle these reverts.
- The comments are misleading, which can cause incorrect assumptions when integrating or debugging.

Recommendation:

If the goal is to get a true `Panic(0x11)`, use an actual arithmetic operation that panics (or an `assert(false)` pattern), or change comments to reflect that this is just an opaque `Error(string)` revert.

INFO

Brittle tests: assert exact revert string from mainnet USDC implementation (upgradeable), risking false failures

Location:

`src/DssLitePsm.t.integration.sol:1126-1132`

Description:

`testBuyGem_Revert_WhenPocketHasNoGem()` asserts an exact revert string from mainnet USDC:

```
vm.expectRevert("ERC20: transfer amount exceeds balance");
litePsm.buyGem(address(this), 1);
```

USDC is upgradeable; revert reasons and error formats can change (custom errors, different strings), making this test fail even if the business behavior (revert on insufficient pocket balance) remains correct.

Impact:

- False negatives in the test suite after upstream token upgrades.
- Reduced usefulness of the test as a stable guardrail for PSM business logic.

Recommendation:

Prefer `vm.expectRevert()` without matching the full string, or match a broader error condition that is stable across upgrades (e.g., no revert string or selector-based checks when possible).

INFO

Comment/code mismatch in chug donation tests: comments claim cut unchanged but assertions show cut changes

Locations:

`src/DssLitePsm.t.integration.sol:646-652`

`src/DssLitePsm.t.integration.sol:666-673`

Description:

In the chug edge-case tests, comments claim the `cut` “didn't change” after additional swaps, but the tests immediately assert that `litePsm.cut()` has changed (from 100k ’ 60k in one test, and from 100k ’ 0 in another).

This is a semantic mismatch that can mislead readers about what `cut()` represents (immediately withdrawable Dai fees) vs `fullCut` (total fees including those currently held as gem value).

Evidence:

`testChug_PartialDaiBalance()`:

```
// Still the cut didn't change, however now is partially in gem
assertEq(litePsm.cut(), 60_000 * WAD, ...);
assertEq(_fullCut(), 100_000 * WAD, ...);
```

`testChug_Revert_WhenNoDaiBalance()`:

```
// Still the cut didn't change, however now is pure gem
assertEq(litePsm.cut(), 0, ...);
assertEq(_fullCut(), 100_000 * WAD, ...);
```

Impact:

Documentation-level confusion: `cut()` is the **currently chuggable Dai amount** (bounded by Dai cash), and it *does* change as Dai cash is consumed.

Recommendation:
Update comments to correctly describe which quantity is unchanged (`_fullCut()`), vs which is expected to change (`litePsm.cut()`).

INFO

Comment/code mismatch: srcTout fee applies to step 2 of described scenario, not step 3

Location:

`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:72-85`

Description:

The Phase2 comments describe a flash-loan scenario and claim that enforcing `srcTout > 0` creates "a fee to be paid in step 3". But `srcTout` is the *buy* fee on the **source** PSM (DAI -> gem), which would be paid in **step 2** (selling DAI into `srcPsm`), not in step 3 (selling gems into `dstPsm`).

This is a semantic mismatch between the narrative and actual fee application.

Evidence:

Comment:

```
 *   2. Sell Dai into `srcPsm` to leave it empty.
 *   3. Sell the gems obtained in step 2 into `dstPsm`.
...
 * To prevent that, we enforce that `srcTout > 0`, so there is a fee to be paid in step 3 above
```

But `srcTout` applies to buying gems from `srcPsm` (step 2), not selling gems into `dstPsm` (step 3).

Impact:

Documentation-level error can lead to incorrect threat-model reasoning and misconfiguration (e.g., assuming fees apply on the wrong leg).

Recommendation:

Fix the comment to match fee mechanics (fee is paid when buying gems from `srcPsm`, i.e., step 2).

INFO

DssLitePsmMom.Halt event documentation mismatches actual parameter type/values

Locations:

`src/DssLitePsmMom.sol:53-58`

`src/DssLitePsmMom.sol:32-36`

Description:

The `Halt` event is declared as:

```
event Halt(address indexed psm, Flow indexed what);
```

where `Flow` is an enum with values `SELL`, `BUY`, `BOTH`.

However, the NatSpec comment for `Halt` says:

```
/// @param what The halted flow. ["tin", "tout"].
```

This is semantically inaccurate: the emitted `what` is not a `bytes32` selector like `"tin"/"tout"`, and it can represent `BOTH` as well.

Impact:

Off-chain consumers, dashboards, or incident tooling that rely on NatSpec / ABI docs can misinterpret event meaning.

Recommendation:

Update the event documentation to reflect the actual enum semantics (SELL/BUY/BOTH) or change the event parameter type to match the documented values.

INFO

DssLitePsmMom.halt() can target any address implementing the interface (limited arbitrary-call router / footgun)

Location:

`src/DssLitePsmMom.sol:123-135`

Description:

`DssLitePsmMom.halt(psm, what)` is `auth`-gated on the Mom contract, but it accepts an arbitrary `psm` address and then performs external calls against it:

```
uint256 halted = DssLitePsmLike(psm).HALTED();
...
DssLitePsmLike(psm).file("tin", halted);
DssLitePsmLike(psm).file("tout", halted);
```

This means any caller authorized by `owner/authority` can use the Mom as a **limited call router** into *any* contract that implements `HALTED()` and `file(bytes32,uint256)` (the callee can interpret those arguments however it wants).

Impact:

Not directly exploitable permissionlessly (requires Mom authorization), but increases governance/configuration risk:

- A broadly-permissioned `authority.canCall` for `halt()` effectively grants the same actor the ability to invoke `file("tin"/"tout", …)` on arbitrary targets.
- Mis-targeting `psm` can halt the wrong LitePSM instance or interact with an unintended contract.

Recommendation:

If the Mom is intended to manage only specific LitePSM instances, consider:

- whitelisting allowed `psm` addresses (set via timelocked governance), or
- deploying a dedicated Mom per LitePSM instance, or
- constraining the `authority` policy to only allow halting intended instances off-chain/on-chain.

INFO

Global debt ceiling headroom in `rush()` can be griefed by increasing system-wide `vat.debt()`, potentially blocking `fill()` during tight ceiling conditions

Location:

`src/DssLitePsm.sol:463-475`

Description:

`fill()` mints Dai based on `rush()`, which is capped not only by the LitePSM's own parameters (`buf`, ilk `line`) but also by **system-wide global debt ceiling headroom**:

```
// src/DssLitePsm.sol
wad = ... _subcap(vat.Line(), vat.debt()) / RAY;
```

Because `vat.debt()` is affected by *any* Maker debt position, an economically motivated attacker can (in principle) open/expand vault debt elsewhere to reduce global headroom and force `rush()==0`, making `fill()` revert (`nothing-to-fill`).

This is an economic/grief vector that becomes relevant when Maker is close to its global debt ceiling: blocking LitePSM refills reduces swap liquidity and can support a DAI premium (benefiting attackers who are long DAI or otherwise profit from reduced DAI issuance).

Impact:

- **Liquidity DoS under stress:** when global headroom is small, a relatively smaller incremental `vat.debt()` increase can block `fill()`, preventing LitePSM from replenishing Dai liquidity.
- **Economic amplification:** can prolong a DAI premium and increase revert risk for large `sellGem` flows.

Recommendation:

- Document this explicitly as an operational limitation: LitePSM liquidity depends on global Maker headroom.
- Consider monitoring/automation that prioritizes global line headroom management during stress events to prevent adversarial blocking.

- (If acceptable) allow an emergency governance path to temporarily raise global headroom or otherwise disable reliance on `fill()` during extreme conditions.

INFO

Misleading 'mints' comments: tests use GodMode.setBalance (does not mint/adjust totalSupply)

Locations:

`src/DssLitePsm.t.integration.sol:145-151`

`src/DssLitePsm.t.integration.sol:1116-1121`

Description:

The test setup comments claim it “mints” `gem` and Dai into the test contract, but it actually uses `GodMode.setBalance`, which directly writes the token balance storage without performing a real mint and without updating `totalSupply`.

This is a semantic mismatch between comments and behavior and can mislead readers into assuming ERC-20 invariants hold (e.g., `totalSupply == sum(balances)`).

Evidence:

In `setUp()`:

```
// Mints 100_000_000 gem into the test contract.
GodMode.setBalance(address(gem), address(this), _wadToAmt(100_000_000 * WAD));
...
// Mints 100_000_000 Dai into the test contract.
GodMode.setBalance(dss.dai, address(this), 100_000_000 * WAD);
```

And in `_setUpGem()` for USDC:

```
// Mints 100_000_000 gem into the test contract.
GodMode.setBalance(_gem, address(this), 100_000_000 * (10 ** GemLike(_gem).decimals()));
```

Impact:

- Can confuse reviewers and future maintainers about the realism of the test environment.
- May hide issues that would only appear when balances are obtained through real mint/transfer flows.

Recommendation:

Update comments to reflect that balances are force-set, or use a minting mechanism appropriate for the token when realism matters.

INFO

Phase 3 sets legacy PSM fees to 0 (not HALTED); if its line is ever re-enabled, old PSM becomes permanently fee-free

Location:

`src/deployment/phase-3/DssLitePsmMigrationPhase3.sol:77-80`

Description:

Phase 3 finalization sets the legacy PSM fees to zero:

```
DssPsmLike(res.srcPsm).file("tin", 0);
DssPsmLike(res.srcPsm).file("tout", 0);
```

While Phase 3 also sets the legacy ilk line to zero in the same transaction, **if governance later re-enables the legacy ilk line** (accidentally or via configuration drift), the old PSM would again become usable with **0% swap fees**, enabling fee-free arbitrage and undermining fee policy.

Impact:

Potential long-tail **economic leakage** if the legacy PSM is unintentionally reactivated.

Recommendation:

Consider setting legacy PSM flows to a halted value (or otherwise explicitly decommission it) rather than leaving it configured as fee-free.

INFO

DssLitePsmMom can be irreversibly bricked by setting both owner and authority to address(0)

Locations:

`src/DssLitePsmMom.sol:75-85`

`src/DssLitePsmMom.sol:100-112`

`src/DssLitePsmMom.sol:123-135`

Description:

`DssLitePsmMom` authorization is based on `owner` and an optional `authority` contract.

The owner can set `owner` and/or `authority` to `address(0)` with no safeguards. If both are set to `address(0)`, then `isAuthorized()` will return `false` for any external caller, permanently disabling `halt()`.

While this may be an intentional “renounce” capability, it becomes a dangerous life-cycle footgun if the mom is expected to remain usable for emergency halts (or if the mom becomes the only remaining ward on a LitePSM instance).

Relevant Code:

```
function setOwner(address owner_) external onlyOwner {
 owner = owner_;
}

function setAuthority(address authority_) external onlyOwner {
 authority = authority_;
}

function isAuthorized(address src, bytes4 sig) internal view returns (bool) {
 if (src == owner) return true;
 else if (authority == address(0)) return false;
 else return AuthorityLike(authority).canCall(src, address(this), sig);
}
```

Impact:

- Governance can accidentally remove all ability to call `halt()`, reducing incident-response capability.
- In combination with LitePSM ward misconfiguration (e.g., leaving only the mom as a ward), this can make the LitePSM effectively unmanageable.

Recommendation:
If renouncing is not intended, disallow setting `owner` and/or `authority` to `address(0)`. If renouncing is intended, consider a two-step process or an explicit `renounce()` with strong warnings, and ensure LitePSM retains other independent wards for recovery.

INFO

Migration/init scripts do not validate Chainlog lookups return valid non-zero contract addresses

Locations:

`src/deployment/DssLitePsmMigration.sol:109-123`

`src/deployment/phase-1/DssLitePsmMigrationPhase1.sol:92-104`

`src/deployment/phase-3/DssLitePsmMigrationPhase3.sol:60-76`

`src/deployment/DssLitePsmInit.sol:115-128`

Description:

Multiple migration/init steps depend on `chainlog.getAddress(key)` returning a valid contract address (e.g., PSMs, AutoLine, IlkRegistry, Vow, Chief). The code uses the returned value immediately without validating:

- it is non-zero, and/or
- it has contract code (when a contract is required).

If a Chainlog key is missing, removed, or repointed incorrectly (Chainlog is mutable by governance), these calls can fail with **opaque decode reverts** (calling a selector on `address(0)`/EOA returns empty data), making the spell hard to diagnose and potentially blocking migration execution.

This is an external-configuration input validation gap.

Evidence:

Examples:

PSM addresses from keys are not checked:

```
src.psm = dss.chainlog.getAddress(cfg.srcPsmKey);
src.ilk = DssPsmLike(src.psm).ilk();   // decode revert if src.psm is 0/EOA
...
dst.psm = dss.chainlog.getAddress(cfg.dstPsmKey);
dst.ilk = DssLitePsmLike(dst.psm).ilk(); // decode revert if dst.psm is 0/EOA
```

AutoLine address not checked (phase 1/3):

```
AutoLineLike autoLine = AutoLineLike(dss.chainlog.getAddress("MCD_IAM_AUTO_LINE"));
```

IlkRegistry/Vow/Chief addresses not checked (init):

```
DssLitePsmLike(inst.litePsm).file("vow", dss.chainlog.getAddress("MCD_VOW"));
DssLitePsmMomLike(inst.mom).setAuthority(dss.chainlog.getAddress("MCD_ADM"));
IlkRegistryLike reg = IlkRegistryLike(dss.chainlog.getAddress("ILK_REGISTRY"));
```

Impact:

- Migration spells can revert unexpectedly due to missing/mistyped Chainlog keys.
- Failures may be difficult to debug on-chain because the revert reason can be a low-level ABI decode error.

In governance operations, this increases the risk of delayed/failed deployments and the need for emergency rescheduling.

Recommendation:

Add explicit validation after each critical Chainlog lookup:

- `require(addr != address(0), ".../missing-chainlog-key")`
- optionally `require(addr.code.length > 0, ".../not-a-contract")` for contracts.

This turns configuration mistakes into clear, actionable revert reasons.

INFO

No recovery mechanism for ERC-20 tokens (including `gem`) accidentally sent to the LitePsm contract; ETH sent via selfdestruct is also stuck

Location:

`src/DssLitePsm.sol:55-542`

Description:

`DssLitePsm` can receive arbitrary ERC-20 transfers (and can also receive ETH via `SELFDESTRUCT`), but it provides **no admin/user recovery path** to move accidentally sent assets back out.

This is most acute for `gem`: the design intends `gem` to live in the separate `pocket`, but if users (or other protocols) mistakenly transfer `gem` to the `DssLitePsm` contract address, those `gem` become **permanently stranded**, because:

- swaps only move `gem` **from/to `pocket`** (`transferFrom(msg.sender, pocket, ...)` and `transferFrom(pocket, usr, ...)`)
- there is no `sweep`, `rescue`, or `withdraw` function
- `pocket` is immutable, so governance cannot repoint the system to treat the LitePsm's own address as the pocket

Evidence:

The contract exposes only swapping/bookkeeping functions (`sellGem`, `buyGem`, `fill`, `trim`, `chug`) and getters; none allow transferring arbitrary tokens out of `address(this)`.

Impact:

Accidentally transferred tokens (especially `gem`) can be locked forever at `DssLitePsm`. Depending on the deployment and operational practices, this can lead to unrecoverable user funds and operational confusion.

Recommendation:

Add an authenticated recovery/sweep function for non-core tokens (and optionally for `gem`/`dai` under strict conditions), or explicitly prevent accidental deposits (where possible) and document operational safeguards (e.g., never transfer `gem` to the LitePsm address; only to `pocket`).

INFO

No-fee swap whitelist is caller-based and allows whitelisted contracts to provide fee-free swaps to arbitrary users/recipients

Location:

`src/DssLitePsm.sol:165-378`

Description:

`DssLitePsm`'s no-fee swap path is gated solely by `bud[msg.sender]` (`toll` modifier). The whitelisted caller can specify an arbitrary `usr` recipient for the output asset.

This means the whitelist is **not** "per end-user" — it is "per calling address". If governance ever whitelists a contract that is publicly callable (e.g., a router/relayer) or otherwise accessible by untrusted users, *anyone* could route swaps through that contract and effectively receive fee-free swaps, defeating the intended permissioning.

While this may be acceptable by design, it is an access-control footgun because the on-chain check does not enforce that the whitelisted actor is also the beneficiary, and it relies on off-chain governance discipline to only whitelist EOAs or contracts with strict internal access controls.

Evidence:

No-fee functions only check the caller and allow arbitrary `usr`:

```
modifier toll() {
 require(bud[msg.sender] == 1, "DssLitePsm/not-whitelisted");
 _;
}

function sellGemNoFee(address usr, uint256 gemAmt) external toll returns (uint256 daiOutWad)
{
 require(tin != HALTED, "DssLitePsm/sell-gem-halted");
 daiOutWad = _sellGem(usr, gemAmt, 0);
}

function buyGemNoFee(address usr, uint256 gemAmt) external toll returns (uint256 daiInWad) {
 require(tout != HALTED, "DssLitePsm/buy-gem-halted");
 daiInWad = _buyGem(usr, gemAmt, 0);
}
```

Impact:

- **Fee bypass via whitelisted relays**: If a broadly accessible contract is whitelisted, untrusted users can obtain fee-free swaps by swapping through it (the contract can custody user funds then call `buyGemNoFee`/`sellGemNoFee` on their behalf).
- **Protocol revenue loss**: Fee revenue (`tin`/`tout`) can be systematically avoided under such a mis-whitelisting.

Recommendation:

- Consider constraining the no-fee path to the caller as beneficiary (e.g., require `usr == msg.sender`) if compatible with intended integrations.
- If arbitrary `usr` is required, explicitly document that only tightly-controlled EOAs/contracts should be whitelisted, and consider monitoring for whitelisted contracts that expose public swap forwarding.

INFO

PSM assumes exact ERC20 transfers (no SafeERC20 / balance-delta checks): fee-on-transfer or false-return tokens can cause fund loss

Locations:

`src/DssLitePsm.sol:336-352`

`src/DssLitePsm.sol:387-399`

`src/deployment/DssLitePsmMigration.sol:169-177`

Description:

The swap logic assumes that: 1) `transferFrom/transfer` always succeed (revert on failure), and 2) the *requested* transfer amount equals the *actual* received amount.

However, the code:

- does **not** use `SafeERC20` / return-value checks, and
- does **not** use balance-before/balance-after accounting to verify the amount received.

This creates loss/drain vectors if the configured `gem` or `dai` token is non-standard:

- **Fee-on-transfer / deflationary tokens:**
  - `sellGem`: pays out Dai based on `gemAmt`, but pocket may receive **less** than `gemAmt`.

The pool can be drained of Dai while receiving insufficient collateral.

  - `buyGem`: transfers Dai in first, then transfers gems out. If Dai is fee-on-transfer, the PSM may receive **less** Dai than `daiInWad` but still releases the full `gemAmt`.
- **ERC20 that returns `false` instead of reverting:** since return values are ignored (interface functions return nothing), a failed transfer can be treated as success, causing either free payout or user loss depending on direction.

The migration code has the same class of issues: it approves and sells tokens without safe wrappers, so a non-standard token can break migrations or (worse) cause incorrect asset movement.

Vulnerable Code:

PSM swaps:

```
// sellGem
 gem.transferFrom(msg.sender, pocket, gemAmt);
 dai.transfer(usr, daiOutWad);

// buyGem
 dai.transferFrom(msg.sender, address(this), daiInWad);
 gem.transferFrom(pocket, usr, gemAmt);
```

Migration sell into dst PSM:

```
GemLike(dst.gem).approve(dst.psm, srcGemAmt);
uint256 daiOutWad = DssLitePsmLike(dst.psm).sellGemNoFee(address(this), srcGemAmt);
```

Impact:

If a non-standard token is configured (or becomes non-standard via upgrade):

- **Direct loss of Dai liquidity** from the PSM (attackers can drain Dai by selling fee-on-transfer gem).
- **Loss of gem reserves** (if Dai is fee-on-transfer, attackers can buy gems while paying less Dai than expected).
- **Broken migrations / unexpected state transitions** during governance operations.

Recommendation:

- Use `SafeERC20` (or equivalent) to handle both non-returning and `false`-returning tokens safely.
- For swap accounting, enforce exact-amount semantics using balance-delta checks:
  - On `sellGem`, measure actual `gem` received by `pocket` (or by the PSM then forward), and compute Dai out from that.
  - On `buyGem`, measure actual Dai received.
- Alternatively, explicitly restrict supported tokens (and enforce it in deployment scripts / runtime checks) to known-safe implementations (no fee-on-transfer, no rebasing, no upgradeable-token risk).

INFO

PSM constructor grants deployer full admin (wards) until an external handoff; misdeployment leaves unintended permanent privileged admin

Locations:

`src/DssLitePsm.sol:176-191`

`src/deployment/DssLitePsmDeploy.sol:33-39`

`lib/dss-test/src/ScriptTools.sol:244-249`

Description:

`DssLitePsm` grants `wards[msg.sender] = 1` in the constructor. The intended owner (e.g., `MCD_PAUSE_PROXY`) is not set in the constructor; instead, the deployment library calls `ScriptTools.switchOwner()` to `rely(newOwner)` and `deny(deployer)`.

This creates a deployment-footgun / access-control hazard:

- If the contract is deployed outside the provided deploy flow, or the post-deploy handoff is skipped/failed/mis-ordered, the deployer remains a permanent admin (ward), able to change fees (`tin/tout`), redirect `vow`, manage the no-fee whitelist (`kiss/diss`), and manage other wards.

Even when using the deploy library, the security model depends on the correctness and atomic execution of the handoff logic.

Evidence:

Constructor grants deployer ward:

```
wards[msg.sender] = 1;
emit Rely(msg.sender);
```

Deployment handoff is external to constructor:

```
ScriptTools.switchOwner(r.litePsm, p.deployer, p.owner);
```

Impact:

If misdeployed, an unintended address retains governance-level privileges over the PSM (high-impact configuration control).

Recommendation:

Make the intended owner an explicit constructor parameter (or otherwise enforce/verify that the deployer ward is removed as part of a single atomic deployment flow). Add defensive checks in deployment/migration scripts to ensure no unexpected wards remain.

INFO

Phase 1 integration test assumes exactly dstWant is migrated, ignoring srcKeep/srcInk/srcArt caps; brittle and numerically incorrect across fork states

Locations:

`src/deployment/phase-1/DssLitePsmMigrationPhase1.t.integration.sol:203-213`

`src/deployment/phase-1/DssLitePsmMigrationPhase1.t.integration.sol:224-228`

Description:

`DssLitePsmMigrationPhase1.t.integration.sol` asserts that the migration always moves exactly `migCfg.dstWant` and that both `srcInk` and `srcArt` decrease by exactly that amount.

However, the underlying migration logic (in `DssLitePsmMigration.migrate`) caps the migrated amount to:

- `min(dstWant, max(0, srcInk - srcKeep))` for collateral moved, and
- `min(srcArt, movedAmount)` for debt wiped.

Therefore, if (at the fork block used by the test runner) any of these conditions hold:

- `srcInk < srcKeep + dstWant`, or
- `srcArt < movedAmount`,

then the actual migration result differs from the test's expectations.

This makes the test **numerically incorrect in general** and potentially **fork-state dependent** (it may pass on one mainnet state and fail on another without any code change).

Evidence:

```
// Old PSM state is set correctly
(uint256 srcInk, uint256 srcArt) = dss.vat.urns(SRC_ILK, address(srcPsm));
assertEq(srcInk, psrcInk - migCfg.dstWant, "after: src ink is not decreased by dst want");
assertEq(srcArt, psrcArt - migCfg.dstWant, "after: src art is not decreased by dst want");
...
assertEq(_amtToWad(gem.balanceOf(address(pocket))), migCfg.dstWant, "after: invalid gem balance for dst pocket");
```

Impact:

- **Brittle integration test**: can fail across fork blocks / evolving mainnet state.
- **False confidence**: the test does not actually verify the migration's capped/min behavior.

Recommendation:

Compute the expected migrated amount using the same min/subcap logic as the migration code (as done in `DssLitePsmMigration.t.integration.sol`), and assert against that value instead of assuming `dstWant` is always fully migrated.

INFO

Phase 1 migration does not validate that the requested `dstWant` amount was actually migrated

Locations:

`src/deployment/phase-1/DssLitePsmMigrationPhase1.sol:61-113`

`src/deployment/DssLitePsmMigration.sol:146-192`

Description:

`DssLitePsmMigrationPhase1.initAndMigrate()` accepts a config value `cfg.dstWant` that (per `MigrationConfig`) represents the maximum amount of gems to move. The phase-1 flow implicitly assumes it has successfully moved `cfg.dstWant` worth of collateral to the new Lite PSM pocket, but it **never checks** how much was actually migrated.

`DssLitePsmMigration.migrate()` computes the migrated amount as:

- `mink = min(cfg.dstWant, max(src.ink - cfg.srcKeep, 0))`

So if the source PSM has less available collateral than expected (e.g., `src.ink` dropped below `cfg.srcKeep + cfg.dstWant` due to pre-migration user activity), the migration will silently move **less than `cfg.dstWant`**, yet phase-1 continues to configure AutoLine and set `buf`, potentially resulting in a Lite PSM with **insufficient pocket liquidity** compared to the intended initial state.

This is a missing postcondition check / state precondition validation.

Evidence:

Phase 1 ignores the migrated amount (`res.sap`) and performs no check:

```
MigrationResult memory res = DssLitePsmMigration.migrate(
 dss,
 MigrationConfig({
 srcPsmKey: cfg.srcPsmKey,
 dstPsmKey: cfg.dstPsmKey,
 srcKeep: cfg.srcKeep,
 dstWant: cfg.dstWant
 })
);
// no require(res.sap == cfg.dstWant) or similar
```

The migration library explicitly allows partial migration:

```
uint256 mink = _min(cfg.dstWant, _subcap(src.ink, cfg.srcKeep));
...
res.sap = mink;
```

Impact:

Operationally, phase-1 can succeed while leaving the Lite PSM underfunded (pocket has fewer gems than intended), which can:

- Reduce or eliminate immediate buy-side liquidity (`buyGem` may revert due to lack of gems in `pocket`).
- Cause downstream governance assumptions (that a specific amount was migrated) to be wrong.

Because this is a migration spell path, failures here are costly to recover from and can require additional governance actions.

Recommendation:

Add an explicit postcondition check in phase-1 (e.g., require migrated amount equals the intended amount when that is the goal, or require a minimum migrated amount), and/or validate preconditions on `src.ink` before running migration so the spell fails fast with a clear message.

INFO

Phase 1/3 migrations can revert inside DssAutoLine.exec due to underflow if global Line < current ilk line (invariant not enforced by Vat)

Locations:

`src/deployment/phase-1/DssLitePsmMigrationPhase1.sol:95-103`

`src/deployment/phase-3/DssLitePsmMigrationPhase3.sol:71-76`

Description:

Both Phase 1 and Phase 3 migrations call `DssAutoLine.exec(ilk)` immediately after updating an ilk's AutoLine parameters.

Maker's `DssAutoLine.exec` updates the **global** debt ceiling with:

```
vat.file("Line", add(sub(vat.Line(), line), lineNew));
```

where `line` is the **current** `vat.ilks(ilk).line`.

Because Maker's `Vat` does **not** enforce `Vat.Line() >= vat.ilks(ilk).line` (or any sum-of-lines invariant), it is possible for governance to set the global ceiling below a specific ilk ceiling. In that state, `sub(vat.Line(), line)` underflows and `exec()` reverts.

As a result, the phase migration spells can become **unexecutable** purely due to a plausible configuration/state mismatch (especially relevant in Phase 3, where the script also manually adjusts the global Line before calling `exec(dstIlk)`, potentially making `Vat.Line()` smaller than the destination's current `line`).

Evidence:

Phase scripts calling `exec`:

```
// Phase 1
autoLine.setIlk(res.srcIlk, cfg.srcMaxLine, cfg.srcGap, cfg.srcTtl);
autoLine.exec(res.srcIlk);
...
autoLine.setIlk(res.dstIlk, cfg.dstMaxLine, cfg.dstGap, cfg.dstTtl);
autoLine.exec(res.dstIlk);

// Phase 3
autoLine.setIlk(res.dstIlk, cfg.dstMaxLine, cfg.dstGap, cfg.dstTtl);
autoLine.exec(res.dstIlk);
```

Upstream Maker `DssAutoLine.exec` arithmetic (external dependency):

From `makerdao/dss-auto-line`:

```
// Set general debt ceiling
vat.file("Line", add(sub(vat.Line(), line), lineNew));
```

Impact:

- **DoS of migrations** (Phase 1 and/or Phase 3) if the global debt ceiling is configured below the relevant ilk's current ceiling.
- This can block governance execution at the exact time migrations are needed.

Recommendation:

Before calling `exec(ilk)`, guard against the missing invariant (e.g., ensure `Vat.Line() >= currentIlkLine`), or adjust the global Line into a safe range before triggering AutoLine updates.

INFO

Phase 2 does not verify dstPsm is actually filled to dstBuf; spell can succeed while leaving no immediate liquidity

Location:

`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:139-145`

Description:

Phase 2 intends to ensure “liquidity available immediately” by calling `fill()` on the destination LitePSM, but the implementation is **best-effort** and does not verify that the buffer target was reached.

If `rush()` is `0` (e.g., because local/global debt ceilings are tight, `dstGap`/`dstMaxLine` are misconfigured, or the LitePSM’s internal accounting deems it cannot mint more), the spell will silently skip `fill()` and still complete successfully.

This is a business-logic gap between intent (immediate post-migration liquidity) and enforcement.

Evidence:

```
// 4. Fill `dstPsm` so there is liquidity available immediately.
if (DssLitePsmLike(res.dstPsm).rush() > 0) {
 DssLitePsmLike(res.dstPsm).fill();
}
```

No post-condition is checked (e.g., destination DAI balance >= `cfg.dstBuf`).

Impact:

- The phase-2 migration can complete with **insufficient** destination DAI liquidity even though `dstBuf` was set.
- Operationally this can break the “immediate liquidity” expectation and require follow-up actions (additional ceiling changes, repeated fills, or separate spells).

Recommendation:

After attempting to fill, enforce a clear post-condition aligned with the migration goal (or deliberately document that phase 2 is best-effort and may leave `dstPsm` underfilled under ceiling constraints).

INFO

DssLitePsmMom halt path can be bricked by misconfigured `authority` (no interface/behavior validation)

Locations:

`src/DssLitePsmMom.sol:65-85`

`src/DssLitePsmMom.sol:109-112`

`src/DssLitePsmMom.sol:123-135`

`src/deployment/DssLitePsmInit.sol:120-125`

Description:

`DssLitePsmMom.halt()` is meant to be callable quickly by governance-appointed actors (via `authority.canCall`) to halt PSM inflow/outflow without the governance delay.

However, `setAuthority()` accepts any address and `isAuthorized()` unconditionally performs an external call to `AuthorityLike(authority).canCall(...)`.

If `authority` is set to an address that:

- does not implement `canCall(address,address,bytes4)` (wrong contract), **or**
- implements it but can revert / consume excessive gas under some conditions,

then **all non-owner callers** will be unable to call `halt()`. In the intended deployment, the owner is typically the PauseProxy (governance), so this misconfiguration can effectively remove the “no-delay emergency halt” capability until a governance action fixes the authority.

This is a *stuck emergency control* risk: the system can be left unable to execute a critical state transition (halting swaps) when needed.

Evidence:

Authorization path:

```
function isAuthorized(address src, bytes4 sig) internal view returns (bool) {
 ...
 } else if (authority == address(0)) {
 return false;
 } else {
 return AuthorityLike(authority).canCall(src, address(this), sig);
 }
}

modifier auth() {
 require(isAuthorized(msg.sender, msg.sig), "DssLitePsmMom/not-authorized");
 _;
}

function setAuthority(address authority_) external onlyOwner {
 authority = authority_;
}

function halt(address psm, Flow what) external auth { ... }
```

Deployment wiring sets `authority` based on chainlog lookups (still externally configurable governance data):

```
// DssLitePsmInit.init
DssLitePsmMomLike(inst.mom).setAuthority(dss.chainlog.getAddress("MCD_ADM"));
```

Impact:

- **High impact** during incidents: authorized actors (e.g., the Hat) may be unable to halt swaps quickly.
- In worst case, this can allow continued swapping while an exploit/market dislocation is ongoing, increasing potential losses.

Recommendation:

Add defensive validation in `setAuthority()` (and/or in `halt`) to reduce the risk of bricking emergency controls, e.g.:

- verify `authority_` supports `canCall` (ERC-165 not available; could use a low-level `staticcall` probe and require success), and
- consider a “break-glass” additional auth path that does not rely on an external `canCall` call.

INFO

Phase 2 migration does not validate src PSM fee range; misconfig can brick old PSM swaps

Locations:

`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:70-86`

`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:135-138`

Description:

`DssLitePsmMigrationPhase2.migrate()` requires `cfg.srcTin > 0` and `cfg.srcTout > 0`, but **does not enforce the documented WAD fee bounds** (0–1e18 where 1e18=100%). It then applies these values directly to the legacy Maker `DssPsm` via `file("tin", ...)` / `file("tout", ...)`.

Maker's canonical `DssPsm` implementation does **not** range-check `tin`/`tout` in `file()`. Values greater than `1e18` can cause arithmetic underflow/overflow inside `sellGem`/`buyGem`, making swaps revert and effectively halting the old PSM.

This is a business-logic/assumption bug: the phase-2 spell is attempting to set *final* parameters safely, but a simple config mistake (or an unexpectedly large value) can permanently DoS a core swap venue.

Evidence:

Phase-2 only checks non-zero:

```
require(cfg.srcTin > 0, ".../src-tin-is-zero");
require(cfg.srcTout > 0, ".../src-tout-is-zero");
...
DssPsmLike(res.srcPsm).file("tin", cfg.srcTin);
DssPsmLike(res.srcPsm).file("tout", cfg.srcTout);
```

The config comments explicitly document the intended scale:

- `srcTin` / `srcTout`: `[wad] - 10**18 = 100%`

…but no upper bound is enforced here.

Impact:

- Governance misconfiguration can set `tin`/`tout` above 100%, causing `DssPsm` swaps to revert.
- This can materially disrupt peg operations and liquidity routing during/after migration.

Recommendation:

Enforce `cfg.srcTin` and `cfg.srcTout` are within the intended range (e.g., `0 < x <= 1e18`) before calling `file()`.

INFO

Phase 3 can become unexecutable if global Vat.Line is below (srcLine + dstLine): Auto-Line.exec reverts after manual Line reduction

Location:

`src/deployment/phase-3/DssLitePsmMigrationPhase3.sol:66-76`

Description:

Phase 3 manually reduces the global debt ceiling by `srcLine`:

```
(,,, uint256 srcLine,) = dss.vat.ilks(res.srcIlk);
dss.vat.file(res.srcIlk, "line", 0);
dss.vat.file("Line", dss.vat.Line() - srcLine);
```

It then calls `DssAutoLine.exec(res.dstIlk)` to (potentially) adjust the destination ilk ceiling and the global ceiling:

```
autoLine.setIlk(res.dstIlk, cfg.dstMaxLine, cfg.dstGap, cfg.dstTtl);
autoLine.exec(res.dstIlk);
```

Maker's `DssAutoLine.exec()` updates the global line using the pattern: `vat.file("Line", (vat.Line() - currentIlkLine) + newIlkLine)`. This **reverts** if `vat.Line() < currentIlkLine` (because it subtracts `currentIlkLine` from `vat.Line()` with checked arithmetic).

Therefore, after Phase 3 reduces `vat.Line` by `srcLine`, the subsequent `auto-Line.exec(dstIlk)` can revert if the system is in (or temporarily enters) a state where:

- `vat.Line() - srcLine < vat.ilks(dstIlk).line`

Equivalently: `vat.Line() < srcLine + dstLine`.

This is an *unchecked assumption* in Phase 3. While many Maker configurations keep `vat.Line()` consistent with the sum of ilk lines, it is not enforced by Vat itself and can be violated by governance parameter changes.

Impact:

- **Operational / economic DoS:** Phase 3 spell can become unexecutable, delaying the final migration and decommissioning of the legacy PSM.
- Because Phase 3 is permissionlessly executable after `eta`, an adversary holding DAI can also choose to execute it at the first moment it becomes valid if the system happens to be in an unfavorable configuration/state, potentially prolonging a two-PSM regime.

Recommendation:

Before reducing `vat.Line`, ensure it will remain >= the destination ilk's current `line` (and/or raise global Line appropriately first). Alternatively, update the destination ilk line directly without relying on `DssAutoLine.exec()` arithmetic that assumes `vat.Line >= currentIlkLine`.

INFO

Phase 3 migration can revert due to underflow when shrinking global debt ceiling by `srcLine` (user-influencable via AutoLine.exec)

Location:

`src/deployment/phase-3/DssLitePsmMigrationPhase3.sol:63-70`

Description:

`DssLitePsmMigrationPhase3.migrate()` reduces the Vat global debt ceiling `Line` by subtracting the source ilk's `line` (`srcLine`).

This subtraction is done in Solidity 0.8 checked arithmetic and will revert if `srcLine > vat.Line()`.

Because `AutoLine.exec(ilk)` is generally permissionless in Maker deployments, a non-privileged user may be able to increase `srcLine` up to `maxLine` shortly before phase-3 execution (subject to TTL/lastInc rules). If `maxLine` (or the resulting `srcLine`) ever exceeds the current global `Line`, phase 3 becomes **unexecutable** (governance DoS).

Even if `srcLine` cannot be pushed above `Line` in the current configuration, the code relies on a fragile invariant (`vat.Line() >= srcLine`) that is not enforced by the Vat and can be broken by governance misconfiguration.

Evidence:

```
autoLine.remIlk(res.srcIlk);

(,,, uint256 srcLine,) = dss.vat.ilks(res.srcIlk);
dss.vat.file(res.srcIlk, "line", 0);
dss.vat.file("Line", dss.vat.Line() - srcLine);
```

Impact:

- **Governance execution DoS**: scheduled phase-3 spell can revert permanently until ceilings are reconfigured and re-scheduled.
- In practice this can delay the final migration and prolong exposure to the legacy PSM.

Recommendation:
Make the global ceiling update robust to configuration drift:

- Clamp the subtraction (`Line = Line > srcLine ? Line - srcLine : 0`), or
- Explicitly validate `vat.Line() >= srcLine` with a clear revert reason and include operational runbooks to prevent griefing, or
- Update `Line` based on desired target values rather than subtracting a live value that can change via permissionless keepers/users.

INFO

Phase migrations assume AutoLine gap/maxLine can accommodate desired LitePSM buf, but do not validate parameter relationships; fill may not reach buf due to ceilings

Locations:

`src/deployment/phase-1/DssLitePsmMigrationPhase1.sol:92-113`

`src/deployment/phase-3/DssLitePsmMigrationPhase3.sol:71-88`

Description:

Phase 1 and Phase 3 migrations (and similarly phase 2) set AutoLine parameters and immediately set the LitePSM `buf`, then call `fill()` to provision liquidity.

However, LitePSM's `fill()` amount is computed by `rush()`, which is capped by both the per-ilk debt ceiling (`vat.ilks(ilk).line`) and the global ceiling (`vat.Line()`):

- `rush()` is limited by `_subcap(line / RAY, Art)`.
- AutoLine's `exec()` typically sets `line` to approximately `Art * RAY + gap` (capped by `maxLine`).

After the migration moves collateral, `Art` is typically about the collateral value already held (H existing minted Dai), and `fill()` is expected to mint roughly `buf` more.

For `fill()` to be able to mint the full configured `buf` immediately, the migration implicitly relies on numeric relationships such as:

- `gap >= buf * RAY` (so that `line/RAY - Art >= buf`), and
- `maxLine` being large enough to not cap below `(Art + buf) * RAY`.

These relationships are **not validated** in the migration scripts. If governance provides an inconsistent config (unit mistake between wad vs rad is a realistic footgun), the script can successfully set `buf` but still mint less than `buf` (or nothing) because of ceiling constraints.

This contradicts the stated intent "so there is liquidity available immediately" and can leave the LitePSM underfunded right after migration.

Evidence (migration scripts):

```
// Phase 1
autoLine.setIlk(res.dstIlk, cfg.dstMaxLine, cfg.dstGap, cfg.dstTtl);
autoLine.exec(res.dstIlk);
DssLitePsmLike(res.dstPsm).file("buf", cfg.dstBuf);
if (DssLitePsmLike(res.dstPsm).rush() > 0) {
 DssLitePsmLike(res.dstPsm).fill();
}

// Phase 3 (same pattern)
autoLine.setIlk(res.dstIlk, cfg.dstMaxLine, cfg.dstGap, cfg.dstTtl);
autoLine.exec(res.dstIlk);
DssLitePsmLike(res.dstPsm).file("buf", cfg.dstBuf);
if (DssLitePsmLike(res.dstPsm).rush() > 0) {
 DssLitePsmLike(res.dstPsm).fill();
}
```

Impact:

- **Incorrect post-migration liquidity**: LitePSM may end up with Dai balance materially below the configured `buf`.
- **Operational risk**: migrations/tests that assume `dai.balanceOf(litePsm) == buf` can fail on-chain if parameters are mis-scaled.

Recommendation:

Add explicit validation of parameter relationships required for the intended immediate fill behavior (especially `dstGap` vs `dstBuf` scaling and `dstMaxLine` sufficiency), or otherwise make the behavior explicit (e.g., accept partial fill and avoid setting `buf` to an unreachable target).

INFO

Phase2 does not validate AutoLine gap parameters; extreme srcGap/dstGap can overflow AutoLine.exec() and make the spell unexecutable

Location:

`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:122-133`

Description:

Phase2 forwards `cfg.srcGap` and `cfg.dstGap` directly into `DssAutoLine.setIlk()` and then immediately calls `DssAutoLine.exec()` for both ilks.

`DssAutoLine` does not bound `gap`. In `exec()` it computes:

```
uint256 lineNew = min(add(debt, ilkGap), ilkLine);
```

where `add()` reverts on overflow.

If `cfg.srcGap`/`cfg.dstGap` is mistakenly set to an extreme value (e.g., near `type(uint256).max`), `add(debt, gap)` can overflow and revert, causing the Phase2 spell to revert and become unexecutable until reconfigured/rescheduled.

Evidence:

Phase2 blindly forwards gap and calls exec:

```
autoLine.setIlk(res.srcIlk, cfg.srcMaxLine, cfg.srcGap, cfg.srcTtl);
autoLine.exec(res.srcIlk);
...
autoLine.setIlk(res.dstIlk, cfg.dstMaxLine, cfg.dstGap, cfg.dstTtl);
autoLine.exec(res.dstIlk);
```

(`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:126-133`)

Upstream AutoLine overflow check:

From MakerDAO `dss-auto-line`:

```
function add(uint256 x, uint256 y) internal pure returns (uint256 z) {
 require((z = x + y) >= x);
}
...
uint256 lineNew = min(add(debt, ilkGap), ilkLine);
```

Source: https://raw.githubusercontent.com/makerdao/dss-auto-line/master/src/DssAutoLine.sol

Impact:

Low likelihood, medium operational impact:

a parameter-entry mistake can permanently block the spell execution (governance action DoS) until fixed.

Recommendation:

Add sanity bounds for `srcGap`/`dstGap` suitable for the system's expected RAD magnitudes, or at least prevent obviously-dangerous sentinel values (e.g., `type(uint256).max`).

INFO

Phase3 migration sets src ilk debt ceiling to zero without validating no other debt positions exist

Location:

`src/deployment/phase-3/DssLitePsmMigrationPhase3.sol:66-69`

Description:

After migrating the legacy PSM's own Vat position, Phase 3 force-sets the **entire source ilk's** debt ceiling (`line`) to zero:

```
(,,, uint256 srcLine,) = dss.vat.ilks(res.srcIlk);
dss.vat.file(res.srcIlk, "line", 0);
```

This assumes the source ilk is *effectively exclusive* to the legacy PSM (i.e., no other urns/users have outstanding or future intended usage of that ilk). The migration code never verifies that assumption (e.g., checking that total `Art` is zero or that the PSM urn is the only relevant urn).

If any other debt positions exist (even unintentionally), setting the ilk ceiling to 0 can unexpectedly freeze further draws for that ilk and change operational behavior for any remaining users/modules.

Impact:

- **Unexpected system behavior / partial DoS** for any non-PSM users/modules of the source ilk.
- Hard to detect from this script alone because it doesn't assert global or per-ilk invariants beyond the PSM's own urn.

Recommendation:

Before setting `line` to 0, assert that the source ilk has no remaining usage beyond the PSM migration (e.g., `Art == 0` or a documented invariant), or explicitly document the assumption that the ilk is PSM-exclusive.

INFO

Self-reference / misconfiguration: pocket can extract Dai liquidity without transferring gem; gem==DAI breaks swap and enables free drain

Locations:

`src/DssLitePsm.sol:176-191`

`src/DssLitePsm.sol:310-352`

Description:

`DssLitePsm` assumes the `gem` token and the `pocket` address are configured to be distinct from Dai and from normal swap participants. However, there are no constructor-time sanity checks.

Two related misconfigurations create severe, non-obvious failure modes:

1) **`msg.sender == pocket` allows free Dai extraction**

In `_sellGem`, the contract transfers `gem` from `msg.sender` to `pocket` and then transfers Dai out:

```
function _sellGem(address usr, uint256 gemAmt, uint256 tin_) internal returns (uint256
daiOutWad) {
 ...
 gem.transferFrom(msg.sender, pocket, gemAmt);
 dai.transfer(usr, daiOutWad);
}
```

If `msg.sender == pocket`, then `gem.transferFrom(pocket, pocket, gemAmt)` is typically a **self-transfer** that leaves the pocket's gem balance unchanged (standard ERC-20 implementations subtract then add to the same address). Yet the Dai transfer still executes.

This means **whoever controls `pocket` can drain the contract's Dai liquidity** by calling `sellGem(pocket, gemAmt)` (or `sellGem(any, ...)`) without actually paying gem to the system.

2) **`gem == dai` breaks the intended 2-asset swap model and enables "swap" drains with only Dai shuffling**

The constructor sets `dai` from `daiJoin.dai()` and independently accepts a `gem_` parameter, but does not enforce `gem_ != dai`:

```
constructor(bytes32 ilk_, address gem_, address daiJoin_, address pocket_) {
 gem = GemLike(gem_);
 dai = GemLike(daiJoin.dai());
 pocket = pocket_;
 ...
}
```

If `gem == dai` and the pocket is attacker-controlled, `sellGem` degenerates into moving Dai from the contract to the attacker while the "payment" leg is a Dai self-transfer into the attacker's pocket.

Impact:

- **High impact misconfiguration footgun**: if `pocket` is mis-set to an address not fully trusted (or later compromised), it can extract all Dai liquidity without providing gem, breaking the module's core invariant and potentially stealing protocol-minted Dai.
- If `gem == dai`, the contract no longer performs a meaningful peg swap and enables highly counterintuitive flows; in combination with an attacker-controlled pocket, this can become a direct drain of the contract's Dai inventory.

Recommendation:

Add constructor-time sanity checks, at minimum:

- `require(gem_ != daiJoin.dai(), "gem-is-dai")`
- `require(pocket_ != address(0), "pocket-zero")`
- Consider `require(pocket_ != msg.sender, ...)` or stronger operational constraints depending on intended governance model.

Also document explicitly that `pocket` must be a trusted address that will never attempt to call `sellGem` as the source of `gem` (or otherwise constrain that behavior in code).

INFO

Fee rounding enables dust swaps with reduced/zero fees and stepwise effective fee schedule

Locations:

`src/DssLitePsm.sol:275-279`

`src/DssLitePsm.sol:336-345`

`src/DssLitePsm.sol:387-393`

Description:

`DssLitePsm` computes fees using integer division:

- Sell: `fee = daiOutWad * tin / WAD` then `daiOutWad -= fee`.
- Buy: `fee = daiInWad * tout / WAD` then `daiInWad += fee`.

Because the division floors, the fee is **discrete** in 1-wei-of-DAI increments. For sufficiently small swaps (and/or sufficiently small `tin`/`tout`), the fee becomes **0**, effectively allowing no-fee swaps even when fees are configured as nonzero. The contract itself documents this risk in `file()`.

While each single-trade rounding error is bounded (< 1 wei of DAI), an attacker can:

- execute dust-sized swaps that pay **no fee** (or systematically underpay relative to the configured percentage), and
- in extreme configurations (very small `tin`/`tout`, or tokens with low transfer granularity), reduce aggregate fees by splitting volume across many transactions.

Evidence:

From `file()` docs:

```
// Swapping fees may not apply due to rounding errors for small swaps where
// `gemAmt < 10**gem.decimals() / tin` or
// `gemAmt < 10**gem.decimals() / tout`.
```

(`src/DssLitePsm.sol:275-279`)

Sell fee flooring:

```
fee = daiOutWad * tin_ / WAD;
unchecked { daiOutWad -= fee; }
```

(src/DssLitePsm.sol:339-344)

Buy fee flooring:

```
fee = daiInWad * tout_ / WAD;
daiInWad += fee;
```

(src/DssLitePsm.sol:390-393)

Impact:

Primarily **fee revenue leakage** and a mismatch between the configured fee rate and the realized effective fee for small swaps. This can weaken the intended economic incentives at the margin (especially for automated market participants performing many small swaps).

Recommendation:

If the intent is “always charge at least 1 wei of fee when `tin/tout > 0`”, add explicit minimum-fee logic; otherwise, document operational constraints on `tin/tout` and expected minimum swap sizes to keep rounding negligible.

INFO

Swaps allow arbitrary `usr` recipients (including address(0) or the LitePsm itself), enabling accidental burns or irrecoverable token locks

Location:

`src/DssLitePsm.sol:310-399`

Description:

`sellGem*` and `buyGem*` accept an arbitrary `usr` recipient address and forward assets to it without validation.

- `sellGem`: sends Dai to `usr` via `dai.transfer(usr, daiOutWad)`.
- `buyGem`: sends `gem` from the `pocket` to `usr` via `gem.transferFrom(pocket, usr, gemAmt)`.

If a caller passes:

- `usr = address(0)`: depending on the token implementation, this can **burn** Dai/`gem` (some ERC-20s do not revert on transfer to zero) or revert unexpectedly.
- `usr = address(this)` (the LitePsm): bought `gem` is sent to the LitePsm contract address, where it becomes **stuck** (there is no `gem` rescue path and swaps only move `gem` to/from the `pocket`).

Even though this is caller-controlled, it's a common footgun for routers/integrators and can lead to permanent asset loss.

Evidence:

```
// _sellGem
...
dai.transfer(usr, daiOutWad);

// _buyGem
...
gem.transferFrom(pocket, usr, gemAmt);
```

(`src/DssLitePsm.sol:347-399`)

Impact:

Accidental burns or permanent token lock at `DssLitePsm` due to invalid recipient addresses.

Recommendation:

Add basic recipient validation (at minimum `require(usr != address(0))`), and consider also disallowing `usr == address(this)` to prevent trapping `gem` inside the LitePsm contract.

INFO

Swaps lack user-side price protection, enabling MEV/governance ordering to worsen execution via fee changes or halts

Locations:

`src/DssLitePsm.sol:310-315`

`src/DssLitePsm.sol:361-365`

`src/DssLitePsm.sol:324-327`

`src/DssLitePsm.sol:375-378`

`src/DssLitePsmMom.sol:123-135`

Description:

All swap entrypoints (`sellGem*`, `buyGem*`) execute at the **current** `tin`/`tout` and have **no** user-provided bounds (`minDaiOut`, `maxDaiIn`, deadlines, etc.). This exposes traders to:

1) **Governance/MEV ordering risk**: a pending swap can be included immediately after a fee update (or a mom-triggered halt), materially changing the effective price paid/received relative to the user's expectation at signing time.

2) **Scheduled-change sniping**: in Maker-style governance, time-delayed actions are often *permissionlessly executable* once `eta` is reached. Searchers can choose the exact block ordering (execute the fee change first, then include the user swap), extracting value from users who rely on off-chain quotes.

This is an economic attack surface rather than a correctness revert: the protocol will behave "as coded", but users can be systematically disadvantaged by transaction ordering.

Evidence:

Swaps use the current `tin`/`tout` only:

```
uint256 tin_ = tin;
require(tin_ != HALTED, ...);
daiOutWad = _sellGem(usr, gemAmt, tin_);
```

(`src/DssLitePsm.sol:310-314`)

```
uint256 tout_ = tout;
require(tout_ != HALTED, ...);
daiInWad = _buyGem(usr, gemAmt, tout_);
```

(`src/DssLitePsm.sol:361-365`)

Emergency halts can change executability/pricing without user involvement:

```
DssLitePsmLike(psm).file("tin", halted);
DssLitePsmLike(psm).file("tout", halted);
```

(`src/DssLitePsmMom.sol:126-132`)

Impact:

- Users can receive **less DAI** than expected on `sellGem` or pay **more DAI** than expected on `buyGem` if fees change between signing and execution.
- In the worst case, a swap can become **unexecutable** (halted) after the user broadcasts it, enabling griefing and/or forcing users to resubmit at worse conditions.

Given that `tin/tout` can be set as high as 100% (or to `HALTED`) via governance/admin flows, the magnitude of adverse execution can be large for users using infinite allowances or automated execution.

Recommendation:

Provide swap variants that accept user limits (e.g., `sellGem(usr, gemAmt, minDaiOut)` / `buyGem(usr, gemAmt, maxDaiIn)` plus optional `deadline`), and encourage integrators to use them.

INFO

file("vow", ...) allows setting vow to address(0), permanently disabling chug() until governance fixes it

Locations:

`src/DssLitePsm.sol:264-272`

`src/DssLitePsm.sol:443-452`

Description:

`DssLitePsm.file(bytes32,address)` allows setting `vow` to any address, including `address(0)`, with no validation.

`chug()` requires `vow != address(0)` and will revert otherwise. Therefore, a mistaken governance action setting `vow = address(0)` disables fee forwarding until corrected. If combined with the “no wards left” lifecycle issue, this can become permanent.

Relevant Code:

```
function file(bytes32 what, address data) external auth {
 if (what == "vow") {
 vow = data;
 } else {
 revert("DssLitePsm/file-unrecognized-param");
 }
}

function chug() external returns (uint256 wad) {
 address vow_ = vow;
 require(vow_ != address(0), "DssLitePsm/chug-missing-vow");
 ...
}
```

Impact:

- Operational misconfiguration can temporarily disable `chug()`.
- In combination with ward removal / bricking, misconfiguration can become permanent.

Recommendation:

Validate `vow` in `file()` (e.g., `require(data != address(0))`) unless disabling `chug()` via `vow=0` is an intentional feature (in which case document it explicitly).

INFO

live() reports vat.live(), but swap functions ignore liveness and remain callable during global settlement (vat.live=0)

Locations:

`src/DssLitePsm.sol:303-399`

`src/DssLitePsm.sol:535-541`

Description:

`DssLitePsm.live()` returns `vat.live()`, implying the PSM's operational status mirrors the Vat's global liveness (e.g., after Maker `cage()` global settlement, `vat.live()` becomes 0).

However, **none of the swap functions** (`sellGem`, `buyGem`, and their no-fee variants) check `vat.live()` (or any local `live` flag). As a result, even when `live()` returns 0, users can still:

- Sell `gem` to receive any Dai already held by the contract (draining remaining Dai liquidity)
- Buy `gem` by sending Dai to the contract (moving collateral out of `pocket`)

This is a business-logic inconsistency between the reported status (`live()`) and actual allowed behaviors, and can produce unexpected behavior during global settlement or other Vat shutdown scenarios.

Evidence:

`live()` is just a passthrough:

```
function live() external view returns (uint256) {
 return vat.live();
}
```

Swap functions do not gate on liveness:

```
function sellGem(address usr, uint256 gemAmt) external returns (uint256 daiOutWad) { ... }
function buyGem(address usr, uint256 gemAmt) external returns (uint256 daiInWad) { ... }
```

Impact:

Medium: During Vat shutdown, the PSM can continue moving Dai/gem despite `live()==0`, potentially causing operational surprises, incorrect assumptions by integrations that check `live()`, and unintended value flows during settlement.

Recommendation:

Decide and enforce the intended shutdown semantics:

- Either gate swaps on `vat.live()==1` (or a local `live` that is caged together with Vat),
- Or clearly document that swaps remain possible after Vat cage and ensure `live()` reflects the PSM's own swap-enabled state (e.g., based on `tin/tout != HALTED`).

INFO

rush()/gush() ignore Vat spot (price) despite assuming spot == RAY, so returned amounts may not be mintable/trim-able in practice

Locations:

`src/DssLitePsm.sol:43-54`

`src/DssLitePsm.sol:463-476`

`src/DssLitePsm.sol:482-496`

`src/DssLitePsm.sol:411-420`

Description:

The contract documentation lists as a core assumption:

- "The `spot` price for gem is always 1 (`10**27`)."

However, `rush()` / `gush()`—which are used to compute how much DAI can be minted (`fill()`) or burned (`trim()`)—**do not check the `spot` value at all**, even though `vat.ilks(ilk)` returns it.

This creates a semantic mismatch: `rush()` is presented as "missing Dai that can be filled", but it can return a positive `wad` even when `fill()` would revert because the vault is unsafe under the current `spot` (or because the large preloaded `ink` causes overflow checks to fail if `spot` is higher than expected).

Evidence:

`rush()` destructures `vat.ilks(ilk)` and ignores the 3rd return value (spot):

```
(uint256 Art, uint256 rate,, uint256 line,) = vat.ilks(ilk);
require(rate == RAY, "DssLitePsm/rate-not-RAY");
// no check that spot == RAY
```

`fill()` trusts `rush()` and then calls `vat.frob(...)`:

```
wad = rush();
require(wad > 0, "DssLitePsm/nothing-to-fill");
vat.frob(ilk, address(this), address(0), address(this), 0, _int256(wad));
```

If `spot` has changed from the assumed value, `frob` safety checks may fail (revert) even though `rush()` returned a positive amount.

Impact:

- `rush()`/`gush()` become unreliable operational indicators.
- `fill()` can unexpectedly revert in cases where `rush()` indicates minting is possible.
- In adverse oracle/Spotter configuration changes, core maintenance flows can be DoSed.

Recommendation:

Explicitly verify `spot == RAY` (and any other required invariants) in `rush()`/`gush()` (and/or directly in `fill()`/`trim()`) so the getters reflect what the subsequent state-changing operation can actually do.

INFO

to18ConversionFactor exponent/subtraction can underflow for tokens with decimals > 18 (deployment-time panic)

Locations:

`src/DssLitePsm.sol:176-186`

`src/DssLitePsm.t.integration.sol:164-170`

`src/DssLitePsm.t.integration.sol:1078-1084`

Description:

Both the production contract and the integration test assume `gem.decimals() <= 18` and compute the scaling factor via:

```
to18ConversionFactor = 10 ** (18 - gem.decimals());
```

If `gem.decimals()` is ever greater than 18, `18 - gem.decimals()` underflows and the contract constructor reverts with a Solidity arithmetic panic (0x11). The integration test reproduces the same assumption in `testTo18ConversionFactor()` and helper conversions `_amtToWad/_wadToAmt`.

Impact:

- **Hard deployment failure** (panic revert) for any collateral token with `decimals > 18`.
- Failure mode is a generic arithmetic panic rather than an explicit, self-documenting error, increasing misconfiguration risk.

Recommendation:

Add an explicit precondition (e.g., `require(gem.decimals() <= 18, ...)`) and/or handle `decimals > 18` by supporting down-scaling rather than only up-scaling.

INFO

Flash-loan ‘protection’ only checks srcTin/srcTout > 0; allows trivially small fees contrary to disincentive intent

Location:

`src/deployment/phase-2/DssLitePsmMigrationPhase2.sol:58-86`

Description:

Phase2 attempts to mitigate two described flash-loan scenarios by requiring `cfg.srcTin > 0` and `cfg.srcTout > 0`.

Semantically, the comments claim this makes the attack costly enough to disincentivize it. However, the guard is only `> 0` in WAD units, so governance can supply an arbitrarily tiny non-zero fee (e.g., 1 wei in WAD = 1e-18%), making the “fee” effectively free for large trades.

This is a correctness/assumption mismatch: the code enforces *non-zero*, but the rationale depends on *economically meaningful* fees.

Evidence:

```
require(cfg.srcTin > 0, "DssLitePsmMigrationConfigPhase2/src-tin-is-zero");
...
require(cfg.srcTout > 0, "DssLitePsmMigrationConfigPhase2/src-tout-is-zero");
```

Impact:

If Phase2 is configured with extremely small `srcTin`/`srcTout`, the documented flash-loan scenarios remain economically feasible (depending on MEV and liquidity), undermining the intended deterrence.

Recommendation:

If the intent is to economically deter these scenarios, enforce a minimum fee floor (protocol-defined) rather than merely `> 0`, or explicitly document that the spell relies on governance choosing a sufficiently large fee.

INFO

Governance footgun: last ward can brick the PSM (including permanent HALTED swaps) by denying all admins

Locations:

`src/DssLitePsm.sol:236-239`

`src/DssLitePsm.sol:283-297`

`src/DssLitePsm.sol:310-327`

`src/DssLitePsm.sol:361-378`

Description:

`DssLitePsm` uses a Maker-style `wards` mapping for admin control. Any ward can call `deny(address)` on **any** address (including itself). There is **no guard preventing revoking the last remaining ward**.

Because `file("tin", HALTED)` and/or `file("tout", HALTED)` are valid admin actions, a ward can (accidentally or maliciously): 1) Halt one or both swap directions by setting `tin`/`tout` to `HALTED`. 2) Remove all wards (including itself) via `deny()`.

Once the final ward is removed, there is no way to call `file()` again to clear `HALTED` (or to fix any other configuration such as `vow`/`buf`). If `tin`/`tout` are HALTED at that time, swaps become **permanently disabled**.

Vulnerable Code:

```
function deny(address usr) external auth {
 wards[usr] = 0;
 emit Deny(usr);
}

function file(bytes32 what, uint256 data) external auth {
 if (what == "tin") {
 require(data == HALTED || data <= WAD, "DssLitePsm/tin-out-of-range");
 tin = data;
 } else if (what == "tout") {
 require(data == HALTED || data <= WAD, "DssLitePsm/tout-out-of-range");
 tout = data;
 } else if (what == "buf") {
 buf = data;
 } else {
 revert("DssLitePsm/file-unrecognized-param");
 }
}

function sellGem(...) external returns (...) {
 uint256 tin_ = tin;
 require(tin_ != HALTED, "DssLitePsm/sell-gem-halted");
 ...
}

function buyGem(...) external returns (...) {
 uint256 tout_ = tout;
 require(tout_ != HALTED, "DssLitePsm/buy-gem-halted");
 ...
}
```

Impact:

- **Permanent DoS of swaps** (one-sided or both-sided), with no recovery path via the contract itself.
- **Permanent loss of governance control** over critical params (`vow`, `buf`, fee rates, whitelist), potentially leading to stuck funds or inability to respond to incidents.

Recommendation:

Add lifecycle protections, such as:

- Prevent denying the last ward (track ward count), and/or
- Require a time-lock / two-step process for setting HALTED, and/or
- Provide a recovery mechanism (e.g., immutable emergency admin) consistent with the protocol's governance model.

INFO

Migration assumes GemJoin scaling matches LitePSM decimals-based conversion factor; mismatch can strand collateral in Vat on executor

Location:

`src/deployment/DssLitePsmMigration.sol:129-172`

Description:

`DssLitePsmMigration.migrate()` converts between Vat-wad collateral (`mink`) and ERC-20 gem units (`srcGemAmt`) using `dstPsm.to18ConversionFactor()`, which is derived from `gem.decimals()`.

However, the source PSM's `GemJoin` adapter is the component that actually defines how many Vat-wad units are released per ERC-20 unit on `exit()`. The migration does **not** assert that the GemJoin's scaling matches `to18ConversionFactor`.

If the GemJoin uses a different scaling than `10**(18-decimals)` (misconfiguration, non-standard join adapter, or token with unusual behavior), the `exit()` can release fewer Vat-wad units than were grabbed, leaving residual `vat.gem(srcIlk, address(this))` collateral stuck on the executor address (PauseProxy in spell execution), with no recovery step.

Evidence:

Key steps:

```
uint256 to18ConversionFactor = DssLitePsmLike(dst.psm).to18ConversionFactor();
...
uint256 srcGemAmt = mink / to18ConversionFactor;
GemJoinLike(src.gemJoin).exit(address(this), srcGemAmt);
```

(see `src/deployment/DssLitePsmMigration.sol:130-158`)

Impact:

- Potentially stranded collateral on the executor (`vat.gem` balance) if scaling mismatches.
- Migration behavior becomes dependent on an implicit assumption that may not hold across ilks/adapters.

Recommendation:

Add an explicit sanity check that the GemJoin exit scaling matches the expected `to18ConversionFactor` (or compute scaling directly from the GemJoin), and/or assert post-exit that `vat.gem(srcIlk, address(this))` is zero (or fully accounted for) before completing the migration.

INFO

Migration can be permanently or griefably DOSed if ilk `rate` ever deviates from RAY (permissionless Jug.drip can trigger deviation when fees nonzero)

Location:

`src/deployment/DssLitePsmMigration.sol:140-143`

Description:

`DssLitePsmMigration.migrate()` hard-requires that both the source and destination ilks have `rate == RAY`:

```
require(src.rate == RAY, "DssLitePsmMigration/invalid-src-ilk-rate");
require(dst.rate == RAY, "DssLitePsmMigration/invalid-dst-ilk-rate");
```

This is framed as an assumption ("stability fees should be set to zero"), but it creates a brittle **DoS surface**:

- In Maker, the accumulated rate increases when stability fees are nonzero and `Jug.drip(ilk)` is called.
- `Jug.drip` is widely documented/used as a **permissionless** method (anyone can call it), meaning a non-privileged actor can trigger rate accrual whenever fees are nonzero.
- Once `rate` increases above `RAY`, it generally does not decrease back to `RAY` under normal operation, making this check potentially **permanently** blocking if fees were ever nonzero for either ilk.

Evidence:

- Rate equality requirements: `src/deployment/DssLitePsmMigration.sol:140-143`.
- Maker documentation describes `Jug.drip()` as a public method used to update accumulated rates, and emphasizes that fee update calls are made frequently and can be invoked by anyone.

Impact:

- **Governance-action DoS**: the migration spell can revert at execution time.
- **Griefing vector** (conditional): if either ilk ever has nonzero fees configured, any third party can call `Jug.drip()` before/around spell execution to ensure `rate != RAY` and force the migration to revert.
- **Potential permanent block**: even accidental temporary fee enablement followed by a single `drip()` can make `rate != RAY` indefinitely, preventing this migration logic from ever being used.

Recommendation:

Avoid strict `rate == RAY` gating if the migration can be made robust to `rate >= RAY` by accounting for the current rate, or enforce/verify via governance that fees are immutable-zero for the relevant ilks and cannot be drifted by `drip()` at any point before migration.

INFO

Migration can revert due to missing bounds check on `mink/mart` int256 conversion

Locations:

`src/deployment/DssLitePsmMigration.sol:85-88`

`src/deployment/DssLitePsmMigration.sol:146-153`

Description:

`DssLitePsmMigration.migrate()` converts `mink` and `mart` (uint256) to int256 via `_int256()` and then negates them for `vat.grab()`.

There is no explicit validation that `mink` and `mart` are <= `type(int256).max`. If they exceed it, `_int256()` reverts with a panic-like `ARITHMETIC_ERROR`.

While this is unlikely under normal Maker parameters, it is still an input/state validation gap: extreme `src.ink` values or misconfigured ceilings could make `mink` large enough to break the migration.

Evidence:

```
function _int256(uint256 x) internal pure returns (int256 y) {
 require((y = int256(x)) >= 0, ARITHMETIC_ERROR);
}
...
dss.vat.grab(src.ilk, src.psm, address(this), address(this), -_int256(mink), -_int256(mart));
```

Impact:

Potential DoS of the migration execution (revert) under extreme-but-possible system states, requiring governance rework.

Recommendation:

Add an explicit bound check for `mink`/`mart` (or compute them in int256 space safely) so the revert reason is clear and the parameter space is explicitly constrained.

Advisory to Clients

Cecuro highly recommends scheduling a follow-up security assessment within six months of this report or immediately after any significant modifications to the codebase, whichever occurs first. This practice is essential for preserving the project's security posture and identifying potential vulnerabilities that may arise from code changes or updates.

Disclaimer

The smart contracts submitted for analysis have been reviewed using Cecuro.ai's AI security analysis system, applying industry-standard vulnerability detection patterns and best practices at the time of this report. The findings disclosed in this report reflect the AI system's assessment of the submitted source code.

This report contains no statements or warranties on the identification of all vulnerabilities or the overall security of the code. Any security review methodology may not detect all potential issues, particularly novel attack vectors, complex business logic flaws, or economic exploits. The report covers the code submitted at the specified version and may not be relevant after any modifications.

Do not consider this report as a final and sufficient assessment regarding the security, utility, or bug-free status of the code. We recommend complementing this analysis with additional independent audits and a public bug bounty program to strengthen the security posture of your smart contracts.

Smart contracts are deployed and executed on blockchain platforms. The platform, its programming language, compiler, and other related software may contain vulnerabilities beyond the scope of code-level analysis. Cecuro.ai cannot guarantee the explicit security of the analyzed smart contracts.

This report does not constitute financial, legal, or investment advice. It is not a recommendation to buy, sell, or invest in any token or project, nor an endorsement of any kind. Users should conduct their own independent due diligence.

cecuro

Agentic Smart Contract Auditing