DIEM Relay Security Review

Version 1.0

Conducted by:
Bretzel, Independent Security Researcher

Table of Contents

1 About Bretzel

I am a smart contract security researcher with experience across Sherlock contests, Pashov Audit Group engagements, and private audits (up to 500m TVL) :

- **Sherlock contests (2025)**: Notional Exponent (Rank 10), Burve (Rank 10), BadgerDAO (Rank 23), Peapods (Rank 5), Plaza (Rank 21).
- **Pashov Audit Group (associate auditor, 2025)**: Portfolio Management, StableSwap Hook (Vyper/Solidity), Launchpad on Doppler, Airdrop/Staking/Rewards, Stablecoin.
- **Private engagements (up to 500m TVL)**: Lagoon (vault infrastructure, **~400m TVL**), Amphor (tokenized funds, **up to ~220m TVL**), QuestV2 (gauge vote acquisition).

To explore a comprehensive overview of my expertise and projects, please visit my website. Feel free to connect with me via Telegram or Twitter.

2 Disclaimer

Reviews are a time, resource and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to find as many vulnerabilities as possible. Reviews can show the presence of vulnerabilities **but not their absence**.

3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

Overview

Project Name	diem-relay
Repository	`https://github.com/Figu3/diem-relay`
Commit hash	Started at cb99cfd067f0be44e881e78fcf3d6488e1e00b42
Resolution	See during Findings
Methods	Manual review

Scope

src/libraries/FullMath.sol
src/libraries/OracleLibrary.sol
src/libraries/TickMath.sol
src/DIEMVault.sol
src/RevenueSplitter.sol
src/csDIEM.sol
src/sDIEM.sol

Issues Found

Critical risk	0
High risk	0
Medium risk	1
Low risk	1
Informational	4

5 System overview

The DIEM Relay staking system consists of four core on-chain components:

1. **`DIEMVault`** — a **USDC deposit vault** for borrowers (Phase 1 is deposit-only; no borrower withdrawals). Deposits are credited off-chain by watching `Deposited` events. The vault enforces **reserve segregation** by tracking **`totalDeposits`** (net borrower deposits) separately from **`protocolFees`** (fee skim), and supports pausing `deposit()` in emergencies.
2. **`sDIEM`** — a Synthetix-style staking rewards contract where users **stake DIEM** and earn **streamed USDC rewards** over 24h periods. All DIEM deposited into `sDIEM` is **immediately forward-staked** into the Venice DIEM staking mechanism (staking is built into the DIEM token contract via `IDIEMStaking`). Withdrawals are **asynchronous**: users `requestWithdraw()` (recording a personal timestamp) and later `completeWithdraw()` after a 24h delay, once the contract has enough liquid DIEM (typically after a Venice `unstake()` is claimed).
3. **`csDIEM`** — an ERC-4626 vault where users **deposit DIEM** and receive **csDIEM shares**. The vault compounds by accepting **permissionless DIEM donations** via `donate()`, which increases `totalAssets()` without minting shares, raising share price. Like `sDIEM`, all DIEM is **immediately forward-staked** on Venice. Standard ERC-4626 `withdraw()`/`redeem()` are disabled; users must use the **async request/complete** flow (`requestRedeem()` → wait 24h → `completeRedeem()`), matching Venice's cooldown.
4. **`RevenueSplitter`** — a **permissionless USDC distribution contract**. When it holds enough USDC (above `minDistribution`), anyone can call `distribute()` to split revenue:
    - **sDIEM leg**: transfers USDC to `sDIEM` and calls `sdiem.notifyRewardAmount()` to start/extend a 24h reward stream.
    - **csDIEM leg**: swaps USDC→DIEM via Aerodrome Slipstream CL router using a **TWAP-derived price floor** and `maxSlippageBps`, then approves csDIEM and calls `csdiem.donate(diemReceived)`.

Venice integration is **permissionless** in both `sDIEM` and `csDIEM`: anyone can call `claimFromVenice()`, `initiateVeniceUnstake()`, and `redeployExcess()` subject to state guards. This removes liveness dependence on a single keeper, while still mitigating cooldown-reset griefing by refusing to initiate a new unstake while Venice cooldown is active.

5.1 Core Features

- **Forward-staking by default (no liquid buffer model)**: Unlike earlier designs, both `sDIEM` and `csDIEM` **stake all received DIEM immediately** into Venice via `diemStaking.stake(amount)`. Liquidity for redemptions/withdrawals comes only from matured Venice unstakes that are later claimed.
- **24h asynchronous exit matching Venice**:
    - `sDIEM`: `requestWithdraw(amount)` (records/accumulates request, reduces stake balance immediately) → `initiateVeniceUnstake()` (batched) → after cooldown `claimFromVenice()` → user `completeWithdraw()`.

  - `csDIEM`: `requestRedeem(shares)` burns shares and records owed DIEM assets → `initiateVeniceUnstake()` (batched) → after cooldown `claimFromVenice()` → `completeRedeem()`.

- **Permissionless Venice operations with cooldown-reset guard**: `initiateVeniceUnstake()` is callable by anyone but reverts if `pending != 0 && block.timestamp < cooldownEnd`, preventing third parties from continually resetting the global cooldown window for pending withdrawals/redemptions.
- **Streaming rewards (sDIEM)**: `notifyRewardAmount(reward)` sets a USDC `rewardRate` streaming linearly over 24 hours. If called mid-period, leftover rewards are rolled into the new rate. A solvency check enforces `rewardRate <= usdc.balanceOf(this)/ REWARDS_DURATION`.
- **Compounding via donation (csDIEM)**: `donate(amount)` is permissionless: pulls DIEM then forward-stakes it. Share price rises because `totalAssets()` increases without minting shares.
- **Correct csDIEM share pricing under forward-stake**: `totalAssets()` includes: [`liquid DIEM + Venice staked + Venice pending - totalPendingRedemptions`] Subtracting `totalPendingRedemptions` is critical because the corresponding shares are already burned and the assets are owed to redeemers.
- **Anti-sandwich swap protection (RevenueSplitter)**: USDC→DIEM swaps use a CL TWAP tick (`OracleLibrary.consult`) to compute `twapOut`, then set `amountOutMinimum = twapOut * (1 - maxSlippageBps)`. An optional absolute floor `minDiemPerUsdc` acts as a circuit breaker if TWAP is stale/manipulated.
- **Reserve segregation + fee skimming (DIEMVault)**: On deposit, `fee = amount * feeBps / 10000`, net credited to `borrowerBalance` and `totalDeposits`, fee to `protocolFees`. Admin can only withdraw up to `protocolFees`.

5.2 Privileged actors

- **Admin (`admin`) — trusted operator (EOA or Safe multi-sig depending on deployment)**:
  - **`DIEMVault` admin** (single-step transfer via `setAdmin`):
    * Can `pause()`/`unpause()` deposits
    * Can set `minDeposit` and `feeBps` (capped by `MAX_FEE_BPS = 50%`)
    * Can withdraw **only** accumulated `protocolFees` via `withdrawProtocolFees(to, amount)`
  - **`sDIEM` admin** (two-step transfer `transferAdmin` → `acceptAdmin`):
    * Can pause/unpause staking and reward claiming (`whenNotPaused` gates `stake()` and `claimReward()`/`exit()`)
    * Can set the `operator`
    * Can recover arbitrary ERC-20s **except** DIEM and USDC (`recoverERC20`)
    * Is the sole signer recognized by `isValidSignature()` (EIP-1271), used by Venice to verify control of the staking address
  - **`csDIEM` admin** (two-step transfer `transferAdmin` → `acceptAdmin`):

    * Can pause/unpause deposits (deposits gated; redemption requests are not pause-gated in this code)
    * Can set the `operator` (role exists though core flows are permissionless here)
    * Can recover arbitrary ERC-20s **except** the underlying DIEM (`recoverERC20`)
  - **`RevenueSplitter` admin** (two-step transfer):
    * Can pause/unpause distribution
    * Can tune `sdiemBps`, `minDistribution`, `maxSlippageBps` (capped at 10%), `swapRouter`, `oraclePool`, `twapWindow` (min 5 min), `tickSpacing`, and `minDiemPerUsdc`
    * Can recover arbitrary ERC-20s **except** USDC (`recoverERC20`)

- **Pending Admin (`pendingAdmin`) — admin transfer recipient**: Exists in `sDIEM`, `csDIEM`, and `RevenueSplitter` for two-step rotation. The pending admin's only privileged action is `acceptAdmin()` to finalize role transfer.
- **Operator (`operator`) — reward notifier (sDIEM) / configured role (csDIEM)**:
  - In **`sDIEM`**, the operator is the **only** address allowed to call `notifyRewardAmount()` (reward seeding). Operator cannot pause, recover tokens, or change config.
  - In **`csDIEM`**, operator is stored and mutable by admin, but the provided code does not grant operator-exclusive powers for Venice management (those are permissionless).
- **Users (any address)**:
  - **Borrowers** deposit USDC into `DIEMVault`; balances are tracked on-chain per borrower (`borrowerBalance`) but withdrawals are not supported in Phase 1.
  - **Stakers**:
    * Stake DIEM into `sDIEM` to earn streamed USDC rewards; withdrawals are delayed and require available liquid DIEM.
    * Deposit DIEM into `csDIEM` to receive shares; redemption is delayed via request/complete and value compounds via donations.
  - **Public callers** can trigger operational functions:
    * `sDIEM`/`csDIEM`: `initiateVeniceUnstake()`, `claimFromVenice()`, `redeployExcess()`
    * `RevenueSplitter`: `distribute()` once above threshold

6 Findings

6.1 Medium risk

6.1.1 [M-01] `initiateVeniceUnstake()` can re-lock already-matured Venice cooldowns (cooldown reset), enabling indefinite withdrawal DoS

Severity: *Medium risk*
Impact: High, Likelihood: Low

Description:

`sDIEM.initiateVeniceUnstake()` & `csDIEM.initiateVeniceUnstake()` are permissionless and call Venice/DIEM staking `initiateUnstake(amount)`. In the DIEM staking implementation, **every** call to `initiateUnstake` resets the global `coolDownEnd` to `block.timestamp + cooldownDuration`, even if an earlier cooldown has already matured but has not yet been finalized via `unstake()`.

Current guard in `sDIEM` & `csDIEM`:

```
require(
    pending == 0 || block.timestamp >= cooldownEnd,
    "sDIEM: Venice cooldown active"
);
```

This prevents starting a new cooldown **while active**, but still allows calling `initiateUnstake()` in the state:

- `pending > 0` (there is a cooldown amount), and
- `block.timestamp >= cooldownEnd` (cooldown matured, but not yet claimed)

In that state, calling `initiateUnstake(amount)` will **reset `coolDownEnd` forward**, effectively **re-locking already-matured funds**. Because the staking position is shared, repeatedly doing this can keep `unstake()` perpetually unavailable for `sDIEM`, preventing funds from returning to the contract and blocking user withdrawals.

Impact:

- **Protocol-wide withdrawal DoS:** matured cooldown funds can be re-locked indefinitely, preventing `diemStaking.unstake()` from being callable (in practice) and starving `sDIEM` of liquidity needed for `completeWithdraw()`.
- Attack is **permissionless** and can be performed with minimal size requests, as long as the attacker can keep `totalPendingNotInitiated > 0` around maturity windows.

Recommendation:

1) Fix: enforce claim-first semantics when `pending > 0`

If any cooldown amount exists (`pending > 0`), do not allow `initiateUnstake()` to be called until the matured cooldown is cleared via `unstake()`:

```
function initiateVeniceUnstake() external override nonReentrant {
    uint256 amount = totalPendingNotInitiated;
    require(amount > 0, "sDIEM: nothing to initiate");
```

```
    (, uint256 cooldownEnd, uint256 pending) = diemStaking.stakedInfos(address(this)
        );

    if (pending > 0) {
        require(block.timestamp >= cooldownEnd, "sDIEM: Venice cooldown active");
        // Clear matured cooldown to avoid re-locking it by resetting cooldownEnd.
        diemStaking.unstake();
    }

    totalPendingNotInitiated = 0;
    emit VeniceUnstakeInitiated(msg.sender, amount);

    diemStaking.initiateUnstake(amount);
}
```

2) **Important UX/Docs clarification:**

Even after the fix, withdrawals are not guaranteed to complete in exactly 24 hours

The patch above solves the **DoS-by-reset** issue, but it does **not** change a fundamental batching property of the design:

- Venice cooldown is **per `sDIEM` staking position**, not per user.
- Only one cooldown can be "in flight" at a time (effectively), because starting a new one while `pending > 0` is disallowed (by the fix) to prevent resets.
- Therefore, **some users' withdrawals will necessarily wait longer than 24 hours** depending on when they request relative to the current batch lifecycle.

Assume the recommended fix is live (so cooldown cannot be reset).

- Batch A starts
  `t = 0`: Bob calls `requestWithdraw(100)`
  `t = 1`: Someone calls `initiateVeniceUnstake()` → Venice cooldown for Batch A runs until `t ~24h`.
- Alice requests during an active cooldown
  `t = 12h`: Alice calls `requestWithdraw(50)`.
  Her personal timestamp `requestedAt = 12h`, so her local 24h delay ends at `t = 36h`.
- But Batch A must finish first
  At `t ~ 24h`: Anyone can call `unstake()` (or `initiateVeniceUnstake()` which will `unstake()` first in the patch). This returns liquidity for Batch A only.
  Alice's 50 DIEM is still in `totalPendingNotInitiated` (Batch B has not been initiated yet).
- Batch B can only start after Batch A is claimed
  Earliest `t ~ 24h`: Someone calls `initiateVeniceUnstake()` again, now initiating Batch B cooldown until `t ~ 48h`.
- Outcome
  Even though Alice's *personal* 24h delay ended at `t = 36h`, she likely cannot `completeWithdraw()` until Batch B is claimed and the contract becomes liquid at `t ~ 48h`.

So Alice waits about **36 hours**, not 24, without any attacker involvement —this is an inherent consequence of global batching + non-overlapping cooldowns.

What to document clearly in user-facing docs and/or NatSpec:

- "Withdrawals have a **minimum** delay of 24h, but can take **longer** depending on whether a Venice cooldown is already in progress and whether the next batch has been initiated/claimed."
- "A user may need to wait up to roughly **48h** in common cases (requesting just after a batch was initiated), and potentially longer if keepers do not promptly call `claimFromVenice()` / `initiateVeniceUnstake()`."

Resolution: Pending

6.2 Low risk

6.2.1 [L-01] Undistributed / dust rewards can become permanently stuck in the staking contract (no recovery path)

Severity: *Low risk*
Impact: Low, Likelihood: High

Description:

The staking contract can accumulate **undistributed reward tokens** that are not attributable to stakers and cannot be returned to the system operator (`RevenueSplitter`) : **Rounding dust during `notifyReward` / reward-rate math**

Many reward systems derive a per-second `rewardRate` via integer division:

- (`rewardRate` = `rewardAmount` / `duration`)
- (`dust` = `rewardAmount` - (`rewardRate` * `duration`))

That remainder ("dust") is **not represented** in `rewardPerToken` accounting and therefore remains in the contract's token balance.

Recommendation:

Implement a mechanism that prevents reward tokens from becoming stranded, while preserving the intended "no arbitrary admin sweep" posture.

Compute the distributable amount and return the remainder to `RevenueSplitter` in the same transaction:

```
function notifyRewardAmount(uint256 rewardAmount) external {
    // auth: only RevenueSplitter (operator)

    uint256 duration = rewardsDuration;

    uint256 rewardRate = rewardAmount / duration;
    uint256 dust = rewardAmount - (rewardRate * duration);

    uint256 distributable = rewardAmount - dust;
```

```
    // set rewardRate/periodFinish using ‘distributable‘
    // ...

    if (dust > 0) {
        IERC20(rewardsToken).transfer(msg.sender, dust); // msg.sender ==
            RevenueSplitter
    }
}
```

Resolution: Pending

6.3 Informational

6.3.1 [I-01] `DIEMVault`: Missing operational safety functions & minor hardening (Phase 1)

Description:

`DIEMVault.sol` is intentionally deployed as a Phase 1 **deposit-only** vault, but it currently lacks a few operational safety affordances and small hardening patterns that are commonly expected in custody-style vaults:

1) `withdrawProtocolFees()` does not use `nonReentrant`

`withdrawProtocolFees()` performs an external ERC-20 transfer. Even though it follows CEI (state is updated before the transfer), adding `nonReentrant` is a standard defense-in-depth hardening step—especially if `depositToken` is ever changed or is not strictly a well-behaved USDC.

2) No “rescue token / stuck funds” mechanism

There is no method to recover: - arbitrary ERC-20 tokens accidentally sent to the vault, or - “excess” `depositToken` that may be transferred directly to the vault outside `deposit()`.

Recommendation:

- Add `nonReentrant` to `withdrawProtocolFees()`
  - This is defense-in-depth; keep CEI as-is.
- Add a sweep/rescue function with strict constraints
  - Allow sweeping *non-deposit tokens* to a safe address.
  - Optionally allow sweeping `depositToken` *only for excess* above accounted liabilities

6.3.2 [I-02] Reward top-ups during an active period can extend the “24h” distribution timeline (Synthetix-style behavior)

Description:

The reward streaming/accounting follows a standard **Synthetix-style** pattern where calling `notifyRewardAmount()` **during an active reward period** blends:

- the undistributed remainder of the current stream, with
- the newly added rewards,

and then restarts emissions over a fresh fixed duration (here, 24 hours).

Conceptually, the logic does:

```
uint256 remaining = periodFinish - block.timestamp;
uint256 leftover  = remaining * rewardRate;

uint256 total = newRewardAmount + leftover;

rewardRate   = total / REWARDS_DURATION;
periodFinish = block.timestamp + REWARDS_DURATION; // resets timeline
```

As a result, even if the protocol advertises "rewards are distributed over 24 hours", users are **not guaranteed** that rewards funded at time (t_0) are fully distributed by (t_0 + 24h) if additional rewards are notified before the period ends.

This is a known tradeoff of the mechanism (and commonly acknowledged in Synthetix-derived distributors).

6.3.3 [I-03] Missing configuration validation & allowance hardening for CL router/oracle integration

Description:

`RevenueSplitter` integrates with a Uniswap V3–style concentrated-liquidity (CL) router and a CL pool TWAP oracle (patterns adapted from Uniswap v3-core).

Because the router/oracle parameters are externally configured (e.g., `swapRouter`, `oraclePool`, `tickSpacing`), the current implementation would benefit from additional **defensive validation**.

Recommendation (hardening checklist):

Validate oracle pool configuration:

- Ensure `oraclePool` corresponds to the intended `(USDC, DIEM, tickSpacing)` market.
- If `ICLPool` exposes `token0()`, `token1()`, `tickSpacing()`, enforce:
  - `token0`/`token1` match `{usdc, diem}` (either order)
  - `pool.tickSpacing()== tickSpacing`
- Optionally sanity-check the pool has initialized observations if your oracle requires them (e.g., `slot0()` / observation cardinality patterns).

6.3.4 [I-04] Custom implementations of Slipstream/Uniswap V3 math libraries increase drift risk

Description:

The codebase ships **local re-implementations** of core Uniswap V3 / Aerodrome Slipstream math libraries:

- `libraries/TickMath.sol`
- `libraries/FullMath.sol`

Instead of importing the canonical upstream implementations (e.g., Aerodrome Slipstream), the project maintains modified copies. While the *intended* mathematical outcomes are the same, the local versions contain **implementation-level differences** (Solidity pragma, `unchecked` regions, signatures/return values, and casting).