CHAINSECURITY

PUBLIC

Code Assessment
of the Permissionless SLL Withdrawals Smart Contracts

August 17, 2026

Produced for

Spark

Contents

1 Executive Summary

Dear all,

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

SparkDAO implements Permissionless SLL Withdrawals, a permissionless backup withdrawal path for Spark savings vaults that sources liquidity from a whitelisted set of venues to complete a redemption when the vault is illiquid and the off-chain ALM Planner is unavailable.

The most critical subjects covered in our audit are functional correctness, access control, and the precision of the withdrawal accounting. Special attention was paid to the containment of the permissionless entry point: the contract acts as an allocator on the controller, the amounts it draws from the ALMProxy are not checked against the obligations towards the Spark vaults, and the per `(asset, vault)` transfer rate limits are the only technical bound on this path. The security of the system therefore additionally depends on a conservative governance configuration, see Configuration Considerations. Security regarding the aforementioned subjects is high, provided the system is correctly configured and operated. The general subjects covered are documentation, specification, error handling, and testing. In summary, we find that the codebase provides a high level of security if correctly configured and operated.

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

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

Sincerely yours,
ChainSecurity

2 Assessment Overview

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

2.1 Scope

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

Version	Date	Commit Hash	Note
1	14 July 2026	87cddffed2c750b1fd26813323f29809f6e5c815	v1.0.0-beta.0
2	4 August 2026	bb0742c8e7640a21a2d0604db747345ceadc6f2a	v1.0.0-rc.0
3	14 August 2026	d1e7ca71d889041e2c54b7cc4dbb33c1730d64c7	v1.0.0

For the Solidity smart contracts, the compiler version `0.8.34` was chosen.

The in-scope files are the following:

```
src/
├── PermissionlessWithdrawals.sol
├── PermissionlessWithdrawalsDiamondPAU.sol
├── PermissionlessWithdrawalsLegacyPAU.sol
└── interfaces/
    └── IPermissionlessWithdrawals.sol
```

2.1.1 Excluded from scope

Everything not listed above is out of scope. The following supporting components were the subject of previous reviews and are not re-examined here:

- The legacy PAU controller (assumed to be `v1.11.0`) and the diamond PAU controller (`v1.14.0`, the most recent version at the time of writing).
- The diamond PAU wiring, which is expected to follow `diamond-pau-deploy` (`v1.0.0`).
- The Spark vaults (`v1.0.1`).

The test suite (`test/`) and third-party dependencies (`lib/`) are likewise out of scope.

The governance configuration (vault and venue whitelisting, penalties, rate limits, and slippage caps) is set by the trusted admin, assumed correct, and out of scope.

3 System Overview

This system overview describes the latest received version (**Version 3**) of the contracts as defined in the Assessment Overview.

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

3.1 Background

Permissionless SLL Withdrawals is a backup withdrawal path for Spark savings vaults. It lets anyone redeem their own vault shares in a single transaction, even when the vault holds no idle liquidity of its own.

Under normal conditions the vault holds enough liquid assets and a user redeems shares directly. This system exists for the case where it does not. A Spark vault deploys most of its assets into yield venues and keeps only a working buffer liquid, so a withdrawal larger than that buffer has nothing to redeem against and reverts. The Savings Intents system handles this today: a user files an on-chain withdrawal intent, and the off-chain ALM Planner relayer unwinds the required liquidity and fulfills the redemption. That works for as long as the relayer is running. If it goes quiet, the intent never settles, and while the user's funds are never at risk, their shares are stuck with no way to reach the underlying assets.

Permissionless SLL Withdrawals removes that dependency. Anyone can call it, and it unwinds the liquidity on-chain within the same transaction, drawing from a governance-whitelisted set of venues with no operator anywhere in the path, so a large withdrawal clears even with the Planner offline. The one cost is a fixed per-vault penalty on every call, sized to cover force-unwinding ALM liquidity and to keep this route a genuine fallback rather than an everyday shortcut.

3.2 Withdrawal Flow

`permissionlessWithdraw(vault, venue, recipient, shares, minAssetsToRecipient)` is the single entry point. Any caller can invoke it, but only redeems their own shares, which they must have approved to the contract, and always pays the vault's configured penalty. `minAssetsToRecipient` is the caller's own floor on the payout net of the penalty, and the call reverts if the final amount falls short of it. The contract never holds funds between calls.

Liquidity is sourced through the Spark ALM controller and its ALMProxy. Holding the controller's relayer/allocator authority, the contract can cover an illiquid vault by moving the assets needed for the redemption into the vault beforehand. A venue is where the proxy's assets are deployed and from which a shortfall is withdrawn. Three venue types are supported, together covering the vault assets (WETH, USDC, USDT, and PYUSD):

- **Aave** (aTokens): withdrawn from an Aave-style lending pool.
- **ERC4626 vaults**: redeemed for the underlying asset, bounded by a per-venue slippage cap.
- **PSM**: does not withdraw the asset directly. It mints USDS and swaps it to the gem (USDC).

Which venue types are valid for a given vault is fixed by governance, not chosen by the caller.

How much the contract will source is bounded before anything moves. The requested amount is the vault's own valuation of the shares, `assetsRequested = vault.convertToAssets(shares)`, and it must satisfy `assetsRequested * 1e18 <= shares * maxExchangeRate`, so that the implied assets-per-share rate stays within the vault's configured `maxExchangeRate`, otherwise the call reverts. That ceiling doubles as the whitelist: a vault whose `maxExchangeRate` is zero is not enabled for this path at all.

The assets backing a redemption are then sourced layer by layer, each covering only what the previous one cannot:

1. **Vault** (regular case): if the vault already holds enough, it redeems directly with no top-up.
2. **Proxy** (first top-up): the vault's shortfall is transferred from the ALMProxy into the vault.
3. **Venue** (second top-up): any remaining shortfall is withdrawn from the configured venue (Aave, ERC4626, or PSM) into the proxy, then forwarded to the vault.

The full `shares` are then redeemed into the contract. A fixed per-vault penalty is sent to `penaltyRecipient` and the remainder to `recipient`, as long as that remainder is at least `minAssetsToRecipient`.

3.3 Configuration

`DEFAULT_ADMIN_ROLE` controls all configuration and can configure the following:

- **`setVaultConfig`**: set a vault's maximum exchange rate and its penalty. A non-zero `maxExchangeRate` whitelists the vault and at the same time caps the assets per share a caller may request, zero disables the vault.
- **`setVaultVenueType`**: register a venue for a vault as Aave, ERC4626, or PSM, checking at configuration time that the venue's underlying matches the vault asset, or unset the pairing again. A PSM venue must be the PSM the controller uses, with the vault asset matching the controller's USDC.
- **`setVenueMaxSharesInRatio`**: sets the slippage cap for ERC4626 venues. It defaults to zero, so a venue fails closed until an admin sets it.
- **`setPenaltyRecipient`**: update the shared penalty recipient.
- **`recoverETH`** / **`recoverERC20`** / **`recoverERC721`**: sweep stranded assets.

Whitelisting a vault on-chain is not sufficient on its own. The backup path also needs the SLL `LIMIT_ASSET_TRANSFER` rate limit for that vault provisioned separately, otherwise every top-up reverts.

3.4 Implementations

Two Permissionless SLL Withdrawals contracts extend the same base, one per controller version: `PermissionlessWithdrawalsLegacyPAU` for the legacy PAU and `PermissionlessWithdrawalsDiamondPAU` for the diamond PAU. They can run in parallel and differ only in how they reach their controller and how their authority over it is held; all withdrawal logic lives in the base. The controller is immutable, so switching between them is a redeployment, not an upgrade.

The legacy implementation calls the controller directly and holds the controller's `RELAYER` role itself. The diamond implementation instead routes every state-changing controller interaction through an `AdministeredAgent`, which is the holder of `ALLOCATOR_ROLE` on the controller's access controls, and the contract is authorized only as one of that agent's actors. The agent, like the controller's access controls, is fixed at construction.

The backup path functions only while its authority links hold. This is an operating precondition, re-checked on every withdrawal by two guards: `_revertIfControllerProxyMismatch` requires the controller to still be the authorized controller on the ALMProxy, and `_revertIfNotRelayer` requires the contract's own authority over the controller, which is the `RELAYER` role for the legacy implementation and, for the diamond one, both the agent holding `ALLOCATOR_ROLE` and the contract still being an actor of that agent. If any of those links is revoked, every withdrawal reverts.

4 Trust Model

4.1 Roles

- `DEFAULT_ADMIN_ROLE`: Spark governance (`SPARK_PROXY`).
  - Trust level: fully trusted.
  - Permissions: full configuration of the contract (per-vault max exchange rates, which double as the vault whitelist; penalties, which may be zero; venue types; ERC4626 slippage cap; penalty recipient; asset recovery; role management).
  - Worst case: no funds held between calls, so no direct drain. Misconfiguration can brick the backup path or, since the contract fronts allocator authority over the controller with a permissionless entry point, expose controller and ALM/PAU actions to any caller (allowing funds to be extracted). Assumed to configure conservatively.
- Caller: any address, via `permissionlessWithdraw`.
  - Trust level: untrusted.
  - Permissions: redeem only their own approved shares (redeem owner is fixed to `msg.sender`) to any recipient, always paying the penalty, with a self-supplied `minAssetsToRecipient` floor on the payout.
  - Worst case: no theft and no access to others' shares. At most griefing of shared SLL rate limits, bounded by the per-call penalty.

4.2 External Dependencies

Fully trusted, assumed correct, and out of scope (previously reviewed):

- Spark ALM controller and ALMProxy:
  - custody funds and enforce rate limits and slippage.
  - neither the controller nor the proxy checks a requested amount against a genuine obligation towards the vault; the per `(asset, vault)` transfer limit and, per share redeemed, the vault's configured `maxExchangeRate` are the bounds on what the permissionless path can draw from the proxy, alongside timely governance intervention in the window before the consumed budget refills.
  - the contract must hold authority over the controller, and the controller must stay authorized on the ALMProxy, both re-checked on every withdrawal and reverting safely otherwise. For the legacy implementation that authority is the contract's own `RELAYER` role; for the diamond one it is indirect, requiring the `AdministeredAgent` to hold `ALLOCATOR_ROLE` and the contract to still be one of its actors.
  - the contracts are immutable and not upgradeable; the diamond controller's functionality can however change through governance-managed facet registration, the proxy's controller assignment is governance-mutable, and so is the agent's actor set.
- `AdministeredAgent` (diamond implementation only):
  - assumed to gate calls on its actor set and to forward them unmodified to the controller.
  - its admin (`SPARK_PROXY` in the assumed wiring) controls actor membership, so it can enable or disable the backup path independently of this contract's own configuration.
  - it holds `ALLOCATOR_ROLE` itself, shared with any other actor it may have, so that authority is not exclusive to this contract.
- Spark savings vault (ERC4626):
  - implements ERC4626 correctly and is non-rebasing.

    - assets are standard ERC-20s: non-rebasing, non-fee-on-transfer, and without transfer hooks or callbacks; USDT's dormant fee, if enabled, would break the payout split (self-reverting, no theft). PYUSD additionally carries an issuer-controlled address freeze and a global pause; an activated pause blocks every PYUSD transfer and with it the entire spPYUSD withdrawal path, including vault-covered redemptions, until lifted.
- Venues (Aave aTokens, ERC4626 vaults, PSM):
    - whitelisted, with the underlying validated against the vault asset at configuration time; a PSM venue is additionally pinned to the PSM and USDC the controller itself uses.
    - the PSM gem is USDC (6 decimals) against 18-decimal USDS.
- Governance configuration:
    - whitelisting, max exchange rates, penalties, rate limits, and slippage caps are assumed correct.

5 System Considerations

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

SC1 Configuration Considerations

System Consideration | Version 1

`permissionlessWithdraw` is a permissionless entry point on a contract that acts as an allocator on the controller, so its configuration governs both the impact a caller can have on the PAU and the economics and availability of the path.

The impact on the PAU is bounded by the following, in order of importance.

Primary:

- Transfer limit. The per-`(asset, vault)` `LIMIT_ASSET_TRANSFER` caps how much can be moved from the ALMProxy into a vault, so it is the ceiling on how much of the PAU this contract can ever draw on. The limit has two dimensions: `maxAmount` bounds a single interval, while `slope` governs how fast it replenishes and therefore how much can be drained on a sustained basis over time. It is most effective when both are sized to reflect the backup withdrawals actually expected for the vault, or to smaller reasonable limits at governance's discretion, since a higher limit or faster replenishment widens that ceiling. Because the limits are independent per-`(asset, vault)` buckets, they bound each vault in isolation but place no ceiling on the total that several vaults can draw from the single shared proxy in parallel, so with multiple vaults live they should be sized against the aggregate as well. Nothing on the path validates a requested amount against a genuine obligation of the proxy towards the vault, since the amount is sized by the vault's own `convertToAssets` and honored on the allocator role and this limit alone. The transfer limit is therefore the only technical safeguard on this path, and its `slope` doubles as the reaction window in which governance can intervene before consumed budget refills.
- Vault whitelisting. Whitelisting is the gate on which vaults are reachable. Whitelisting a wrong or malicious vault would let a caller pull otherwise-unrelated PAU funds through it, bounded only by the transfer limit, so its correctness is critical. Whitelisting alone does not make a vault usable: unless the transfer limit is provisioned.

Secondary:

- Venue configuration. Venues are asset-matched at configuration time and are rate-limited, so only legitimate venues are usable and a misconfiguration fails closed, making the path unusable rather than harmful. The PSM venue is the exception to the per-`(asset, vault)` containment: its path consumes the global `usds_mint` and `usds`-to-`usdc` budgets shared with the rest of the ALM system, so its usage contends system-wide in both directions rather than staying vault-local.
- `maxSharesInRatios`. Usually not decisive, but a useful defense-in-depth bound on the shares burned on the ERC4626 path, hardening it against a mispriced or misbehaving ERC4626 venue. Two configuration pitfalls apply: the ratio must encode the share/asset decimal gap (a 6-decimal asset in an 18-decimal-share venue needs an additional `1e12` factor, and mis-encoding silently makes the venue unusable), and the ratio persists after a venue is disabled with `setVenueType(NONE)` (now `setVaultVenueType(UNSET)` as of **Version 2**), so a later re-enable inherits the stale ratio instead of failing closed like a fresh venue.

The penalty parameters instead shape the path's economics and availability.

- Penalty amount. A fixed per-vault amount deducted from every withdrawal, disincentivizing this path in favor of the regular withdrawal routes. Being a constant rather than proportional, it is only economical where it is small relative to the amount withdrawn, so it naturally restricts the path to large withdrawals. It

also imposes a hard floor rather than only an economic one: a withdrawal whose redeemed assets do not cover the penalty reverts, so sub-penalty withdrawals are impossible.

- Penalty recipient. A single address, shared across all vaults and assets, paid the penalty on every withdrawal before the recipient is paid, so it should be a trusted address that is not blacklisted or freezable on any in-scope asset. A blacklisted or frozen recipient would make the penalty transfer revert and temporarily deny the path for that asset until it is changed. USDC's blacklist and PYUSD's freeze both restrict transfers to a listed address and would trigger this; USDT's blacklist is sender-side only and would not.

SC2 Griefing via Liquidity and Rate-Limit Exhaustion

System Consideration | Version 1

`permissionlessWithdraw` enables two griefing vectors that cost the fixed penalty and gas:

- Grief other withdrawals: by consuming the available liquidity, or the relevant withdraw or transfer rate limit, ahead of a victim, the griefer makes the victim's withdrawal revert.
- PAU operations: the same consumption drains the rate-limit budgets and liquidity that the PAU's own operations rely on, stalling them (for finite limits).

Redepositing the proceeds does not sustain the attack: the funds return to the vault as available liquidity.

SC3 Stale Configuration on Re-Whitelisting

System Consideration | Version 1

De-whitelisting a vault and later re-whitelisting it does not fully reset its configuration. `setVaultConfig` updates only the `whitelisted` flag (now `maxExchangeRate` as of **Version 2**) and `penaltyAmount`, so the following state persists across the cycle:

- `venueTypes`: the per-vault venue sub-mapping is never cleared, so previously configured `(vault, venue)` routes are silently re-activated once the vault is whitelisted again. The only way to clear a route is an explicit `setVenueType(vault, venue, VenueType.NONE)` (now `setVaultVenueType(vault, venue, VenueType.UNSET)` as of **Version 2**).
- `_maxSharesInRatios`: the per-venue slippage ratios persist. This is expected and benign, since `_maxSharesInRatios` is a global per-venue parameter rather than vault-scoped.
- `penaltyAmount`: the penalty persists, and `setVaultConfig` requires it to be non-zero even when de-whitelisting (non-zero requirement removed as of **Version 2**).

The admin should be aware of this persistence and review a vault's venue configuration when re-whitelisting it, rather than assuming a clean slate.

SC4 ALM Proxy Outflows Are Not Bounded by Vault Obligations

System Consideration | Version 1

The ALMProxy custodies the funds of the entire ALM system, most of which do not originate from the Spark savings vaults. Until now, funds moved between the proxy and a vault only through privileged actions: the ALM Planner, holding the controller's operational role (`RELAYER` on the legacy controller, `ALLOCATOR_ROLE` on the diamond), initiates transfers and governance configures the system. The permissionless withdrawal path changes this boundary. Any caller can now trigger an outflow from the proxy into any of the whitelisted vaults.

The outflow is sized solely by the vault's own accounting: the contract requests `convertToAssets(shares)` and covers the vault's shortfall with `transferAsset(asset, vault, amount)`. This is the same function through which the ALM Planner returns funds to a vault, both consuming the same per `(asset, vault)` `LIMIT_ASSET_TRANSFER` rate limit, and this rate limit, disconnected from the actual obligations, is the only limit on the outflow. At the time of writing, the live mainnet configuration sets it, along with the take limits, to unlimited for all four vaults; it is expected to be limited prior to the deployment of this contract. No component compares the amount against the proxy's genuine obligations towards the vault, even though these are reconstructible on-chain: both directions of the channel are mediated by single metered functions (`takeFromSparkVault`, respectively the spark vault facet's `take`, towards the proxy, and `transferAsset` back), and the obligations exceed what was taken only by the interest the vault accrues in the meantime.

The savings vaults are upgradeable (UUPS) proxies: whatever a vault's `convertToAssets` reports, now or after any future upgrade, determines what this path pulls from the proxy, with the proceeds delivered to an arbitrary recipient and, where the proxy's idle balance is short, the ALM's venue positions force unwound to cover the transfer. No safeguard exists that is independent of the vaults' accounting.

There is no separation or defense in depth here, contrary to elsewhere in the system. In theory, a vault can take all reachable assets of the ALM proxy, irrespective of the obligations towards it. Even though the vaults are a subsystem of the same protocol, defense in depth exists precisely to limit such a clearly unexpected movement of funds.

SC5 Broken Venue Withdrawals

System Consideration **Version 1**

The venue withdrawal is triggered permissionlessly, so any caller can invoke it directly. In the operator-driven flow the relayer is only lightly trusted, but for withdrawals it is granted more latitude, so that an emergency such as a depeg can be met with a fast exit and a broken withdrawal is tolerated. The permissionless path removes that operator discretion, since any user triggers the withdrawal directly.

The path adds no per-call guard against a broken or lossy venue withdrawal. If a venue withdrawal were buggy so that withdrawing a small amount consumed a disproportionate share of the position (say, withdrawing 10 burns all of the aTokens), the loss would land on the ALM, with no per-call bound on the Aave or PSM paths (the ERC4626 path is bounded by `maxSharesInRatio`). A value-delta check, comparing the change in the venue position against the assets actually received, would bound this as defense in depth.

Such a guard is not added: trust is placed in the venues, which follows from the trust already placed in the controller and governance that whitelist and configure them. The venues are therefore relied upon to withdraw correctly, and no additional per-call protection is layered onto this path.

6 Terminology

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

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

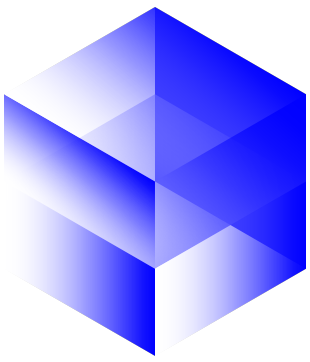
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	**Impact**		
	High	Medium	Low
High	**Critical**	**High**	**Medium**
Medium	**High**	**Medium**	**Low**
Low	**Medium**	**Low**	**Low**

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

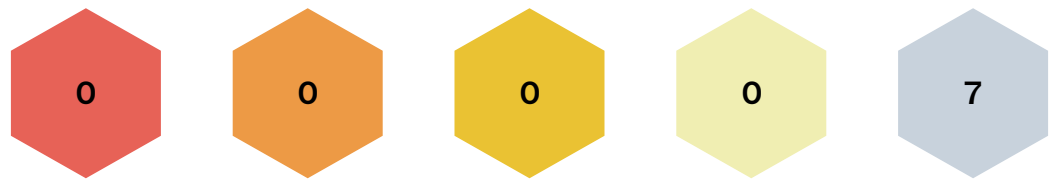
7 Findings

In this section, we describe the findings identified during the course of the engagement. The findings are categorized by severity as explained in the Terminology section. Below we provide an overview of the total number of findings per severity, as well as the number of open issues.



No Open Issues

Total number of findings



Critical | High | Medium | Low | Informational

7.1 Overview

The following table lists all identified findings along with their severity and current status. A finding is considered resolved once it has been fully corrected or the specification has been changed accordingly. Non-informational findings with any other status, including partial fixes, acknowledgements, and accepted risks, remain open.

Info	#001	Documentation and Specification Inaccuracies	Specification Changed ✓
Info	#002	Duplicated Constant	Code Corrected ✓
Info	#003	Missing Events	Code Corrected ✓
Info	#004	Selector Collision Risk in Token Recovery	Code Corrected ✓
Info	#005	Consistency Check Covers Only One Authority Link	Code Corrected ✓
Info	#006	Further Documentation and Interface Specification Drift	Specification Changed ✓
Info	#007	UncheckedAgent Constructor Argument	Code Corrected ✓

7.2 Descriptions

#001 Documentation and Specification Inaccuracies

Informational	Version 1	Specification Changed

Several statements in the documentation, technical specification, and interface NatSpec do not match the implementation. None of them affect on-chain behavior.

1. TECH_DOC step 1 misstates validation in both directions. It claims the `(vault, venue)` pair is validated as whitelisted, but step 1 performs no venue check; the `(vault, venue)` pair's type (the de facto venue whitelist) is only consulted later, in `_withdrawFromVenue`, and only when a shortfall must be sourced. It also omits the checks step 1 actually performs: `shares != 0` (`ZeroShares`) and the controller/proxy consistency check `_revertIfControllerProxyMismatch`. The upfront-validation claim also contradicts TECH_DOC's own Design Choices bullet, which relies on the venue being validated lazily (a misconfigured venue is never touched when no venue draw is needed).
2. TECH_DOC describes a venue asset-match check running "before drawing from a venue" and reverting `IncorrectVenue`. In reality the asset-match check runs at configuration time in `setVenueType` (reverting `IncorrectVenue` there), so an asset-mismatched venue can never be stored; the withdrawal path performs no asset check and instead reverts with `VenueTypeNotSet` when the venue type is unset.
3. TECH_DOC step 3 calls the vault top-up guard a "balance delta check". It is an absolute-balance check `balance >= assetsToTransfer` located in the step 4 top-up block, and it compares against `assetsToTransfer` (the full vault shortfall), not the venue-requested amount.
4. TECH_DOC refers to a "hardcoded set of venues". Venue addresses are admin-configured at runtime via `setVenueType`; only the venue types (the `VenueType` enum and its dispatch logic in `_withdrawFromVenue`) are hardcoded.
5. The interface NatSpec for the `controller()` getter says it "Returns the configured MainnetController address", but the getter is generic; only the legacy implementation targets a MainnetController-style controller, while the diamond implementation uses a different controller ABI. The notice should be generic.
6. README and TECH_DOC list the supported assets as "USDC, USDT, and ETH". The spETH vault's underlying asset is WETH and the flow contains no unwrap step; recipients receive WETH, not native ETH. README additionally labels these assets "venues", conflating them with the actual venues (SparkLend, ERC4626 vaults, the PSM) listed in the preceding paragraph.
7. TECH_DOC states that an enabled USDT fee "would break the exact penalty plus recipient amount split". The actual effect is a revert, not a wrong split: the top-up and/or the redeemed amount arrive short, so either the redeem or the second `safeTransfer` fails and the spUSDT path is unavailable until the fee is disabled.
8. In the architecture diagram (`architecture_flow.png`) the user edges are labeled "Aprove" (typo, twice).
9. README's Architecture paragraph narrates the flow as "the user redeems shares, the contract sources any shortfall from the venue". This is the reverse of the actual (and correctly diagrammed) order: the shortfall is sourced into the vault first, precisely so that the subsequent redeem can succeed.

Specification Changed

The docs have been improved.

#002 Duplicated Constant

Informational	Version 1	Code Corrected

The constant `_USDS_CONVERSION_PRECISION` (value `1e12`) is declared identically in both `PermissionlessWithdrawalsDiamondPAU` and `PermissionlessWithdrawalsLegacyPAU`. This is unnecessary duplication; the constant could be defined once in the shared abstract base `PermissionlessWithdrawals` that both extend.

Code Corrected

The constant has been moved to the abstract contract.

#003 Missing Events

Informational	Version 1	Code Corrected

Not all state-changing actions emit a corresponding event. The constructor sets `penaltyRecipient` without emitting `PenaltyRecipientSet`, unlike the `setPenaltyRecipient` setter which does. Off-chain consumers that track `PenaltyRecipientSet` to learn the current penalty recipient miss the initial value configured at deployment.

Code Corrected

The constructor now emits the event.

#004 Selector Collision Risk in Token Recovery

Informational	Version 1	Code Corrected

`recoverERC20` and `recoverERC721` transfer assets out of the contract for an arbitrary admin-supplied `token`, calling `transfer(address,uint256)` (via `SafeERC20`), `balanceOf(address)`, or `safeTransferFrom(address,address,uint256)` on it, and do not exclude the `controller`. The contract is an allocator/relayer on the `controller`, so if `token` were set to it and it exposed a function whose 4-byte selector matched one of those three, the recovery call would invoke that privileged function instead of moving a token.

No collision exists for the `MainnetController` (`v1.11`) or the diamond PAU (as of `v1.14`, with wiring per `diamond-pau-deploy` `v1.0.0`). Future changes to either could introduce a colliding selector and lead to unexpected behavior. These are privileged, admin-only functions, so it would require the trusted admin to set `token` to the controller.

Code Corrected

`token != controller` is now validated. Similarly, it is ensured that `recipient != controller` for `recoverETH` so that the controller is not invoked.

#005 Consistency Check Covers Only One Authority Link

Informational	Version 1	Code Corrected

`_revertIfControllerProxyMismatch` verifies that the controller still holds the `CONTROLLER` role on the ALMProxy, but not that this contract still holds the `RELAYER` (legacy) or `ALLOCATOR_ROLE` (diamond) role on the controller, which every venue-sourcing withdrawal also requires. If that grant is missing or later revoked, the check still passes and the withdrawal instead reverts deep inside a controller hook with an opaque access-control error rather than failing fast. This is fail-safe (a revert, no fund loss) and the grant is a normal deployment prerequisite, but asserting this contract's own authority on the controller upfront would produce a clearer revert.

> **Code Corrected**
>
> A `_revertIfNotRelayer` check was added and is now called in `permissionlessWithdraw` alongside `_revertIfControllerProxyMismatch`, so a withdrawal fails fast with `NotRelayerOnController` rather than reverting deep inside a controller hook. Each implementation supplies its own `_isRelayer`: the legacy PAU checks that this contract holds the `RELAYER` role on the controller, while the diamond PAU checks that the administered agent holds `ALLOCATOR_ROLE` and that this contract is an active actor of that agent.

#006 Further Documentation and Interface Specification Drift

Informational	Version 2	Specification Changed

Further statements in TECH_DOC, README, and the interface NatSpec do not match the implementation at **Version 2**. None affect on-chain behavior.

1. TECH_DOC documents `permissionlessWithdraw(vault, venue, recipient, shares)`, omitting the fifth parameter `minAssetsToRecipient` (the caller's only slippage guard, enforced by the `InsufficientAssetsForRecipient` revert); step 6 and README omit it too.
2. The interface names that parameter `minAssetsForRecipient`, the implementation `minAssetsToRecipient`.
3. TECH_DOC says `_USDS_CONVERSION_PRECISION` is "defined in each concrete implementation"; it is defined once in the abstract base.
4. TECH_DOC names four virtual hooks and says the implementations implement "only those"; there are seven (also `_getPSM`, `_getPSMUSDC`, `_isRelayer`).
5. The `recoverETH` NatSpec omits its `RecipientIsController` and `TransferETHFailed` reverts, which the `recoverERC20` / `recoverERC721` siblings do document.
6. The `setVaultConfig` NatSpec says it "Reverts if penaltyAmount is zero", but that guard and its `ZeroPenaltyAmount` error were removed in **Version 2**; a zero penalty is now accepted.
7. The Trust Assumptions and Pausing sections say this contract holds `ALLOCATOR_ROLE` and is disabled by revoking it; for the diamond the agent holds the role and the disable lever is `agent.removeActor(...)`. The agent dependency and its constructor argument are undocumented.
8. The decimal dependency of `maxExchangeRate` is undocumented: it must encode `10**(assetDecimals - shareDecimals)` (opposite to `maxSharesInRatio`, which is documented). A wrong-decimals value silently disables the cap.

9. The provisioning checklist omits the venue deposit-side rate-limit keys (`LIMIT_AAVE_DEPOSIT` / `LIMIT_4626_DEPOSIT`); on the legacy controller a zero deposit key reverts every venue withdrawal. Diamond is unaffected.
10. The `IncorrectVenue` NatSpec ("asset does not match the vault's") is inaccurate: the PSM branch reverts on `venue != _getPSM()` regardless of asset, and a mistyped AAVE/ERC4626 venue reverts with empty data instead.

Specification Changed

The docs were mainly corrected, with one minor divergence remaining:

- 7: Trust Assumptions and the constructor argument were corrected, but the Pausing section still attributes the diamond `ALLOCATOR_ROLE` to this contract and omits the `agent.removeActor(...)` disable lever.

#007 Unchecked administeredAgent Constructor Argument

Informational | **Version 2** | **Code Corrected**

The diamond constructor assigns `administeredAgent_` to an immutable without a zero-check, unlike the base constructor's other address arguments; a zero value makes `_isRelayer` permanently false, reverting every `permissionlessWithdraw` with `NotRelayerOnController`.

Code Corrected

A zero-check on `administeredAgent_` has been added.

8 Limitations and use of report

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

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