yAudit

June 2026

Prepared for
Centrifuge

Audited by
adriro
watermelon

Centrifuge v3.3

Smart Contract Security Assessment

Contents

1 Review Summary **2**
1.1 Protocol Overview . . . 2
1.2 Audit Scope . . . 2
1.3 Risk Assessment Framework . . . 3
1.3.1 Severity Classification . . . 3
1.4 Key Findings . . . 4
1.5 Overall Assessment . . . 4

2 Audit Overview **4**
2.1 Project Information . . . 4
2.2 Audit Timeline . . . 4
2.3 Audit Resources . . . 4
2.4 Critical Findings . . . 4
2.5 High Findings . . . 5
2.6 Medium Findings . . . 5
2.6.1 StdManifest prevents routine OracleValuation price updates . . . 5
2.7 Low Findings . . . 6
2.7.1 Permissionless NAV recomputes can refresh the manifest share-price baseline and delay guarded commits . . . 6
2.7.2 `SimplePriceManager` cannot execute timelocked price exceptions . . . 7
2.7.3 `notifyShareClass()` bypasses the timelock for initial share hooks . . . 8
2.8 Gas Savings Findings . . . 9
2.9 Informational Findings . . . 9
2.9.1 Dependency setters can leave `StdManifest` using stale addresses . . . 9
2.9.2 `BridgeCircuitBreaker` does not implement a sliding window rate limiter 10
2.9.3 `StdManifest` does not enforce a longer escalation delay . . . 10
2.9.4 HubRegistry authorization ids are malleable across equivalent calldata . . 11

1 Review Summary

1.1 Protocol Overview

Centrifuge v3.3 builds on top of the protocol's exisiting tokenization functionalities, adding a per-pool security policy enforcement layer which gates every sensitive manager action behind a timelock, providing independent actors a window to veto any malicious action before it is executed. The upgrade also introduces a circuit breaker contract to rate limit the amount of pool shares sent from a selected chain over an arbitrary time window.

1.2 Audit Scope

This audit covers 20 smart contracts across 5 days of review. This was a mix of a full review for newly implemented contracts and a diff review for previously reviewed ones which integrate with the new additions. The auditors focused on PRs 197, 204, 209, 214 and 218.

```
src
├── core
│   ├── hub
│   │   ├── HubHandler.sol
│   │   ├── HubRegistry.sol
│   │   └── Hub.sol
│   ├── messaging
│   │   ├── libraries
│   │   │   └── MessageLib.sol
│   │   ├── MessageDispatcher.sol
│   │   └── MessageProcessor.sol
│   └── utils
│       ├── BatchedMulticall.sol
│       ├── ContractUpdateLib.sol
│       ├── ContractUpdaterForwarder.sol
│       └── Envoy.sol
├── hooks
│   ├── accounting
│   │   └── NAVManager.sol
│   └── bridge
│       └── BridgeCircuitBreaker.sol
├── managers
│   └── hub
│       └── Supervisor.sol
├── manifests
│   └── StdManifest.sol
├── misc
│   ├── libraries
│   │   ├── ArrayLib.sol
│   │   ├── BytesLib.sol
│   │   └── MathLib.sol
│   └── types
│       └── D18.sol
```

```
├── valuations
│   └── OracleValuation.sol
└── vaults
    └── BatchRequestManager.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	3
Informational	4

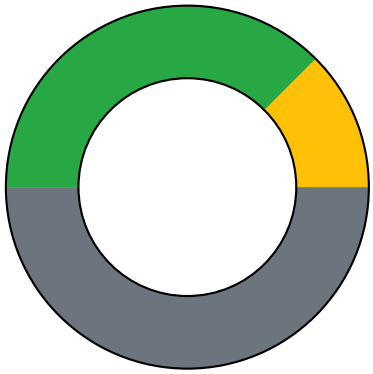

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The upgrade defines a StdManifest contract which implements an opinionated, default policy. The security policy is well designed and implemented, although a self DoS vector was identified for vaults that have opted into the onchain accounting functionality. Some minor documentation improvement suggestions where raised.

2 Audit Overview

2.1 Project Information

Protocol Name: Centrifuge
Repository: https://github.com/centrifuge/protocol-internal/
Commit Hash: 23b3e9666d60c2d014092e4b60e4360017e23509
Final Commit Hash:
Commit URL: https://github.com/centrifuge/protocol-internal/tree/23b3e9666d60c2d014092e4b60e4360017e23509

2.2 Audit Timeline

The audit was conducted from June 6 to 10, 2026.

2.3 Audit Resources

- Code repositories and protocol documentation

2.4 Critical Findings

None.

2.5 High Findings

None.

2.6 Medium Findings

2.6.1 StdManifest prevents routine OracleValuation price updates

`StdManifest` prevents the trusted `OracleValuation` price-feed flow from completing. Pools that install the standard manifest cannot process normal local or remote oracle updates, leaving the holding and its accounting value stale.

Technical Details

Both `OracleValuation.setPrice()` and `OracleValuation.untrustedCall()` route through `_setPrice()`. `_setPrice()` records `newPrice` and immediately calls `Hub.updateHoldingValue()` so the Hub recomputes the holding value and its accounting entries. `Hub.updateHoldingValue()` invokes `_protected()`, which identifies `OracleValuation` as the caller and applies the pool manifest.
When `onchainAccounting` is enabled, `StdManifest._classify()` considers `updateHoldingValue()` an accounting selector and permits only the configured `NAVManager`. The call from `OracleValuation` therefore reverts with `OnchainAccountingOnly`. An authorization cannot make this path executable because the execution-time caller remains `OracleValuation`.
When `onchainAccounting` is disabled, `updateHoldingValue()` reaches the default delayed branch. A manager must pre-authorize the exact `updateHoldingValue(poolId, scId, assetId)` calldata and wait for it to mature before a feeder can submit a price. Without it, the Hub reverts and the outer oracle transaction rolls back its price write.
This authorization approves only a future revaluation of a holding. The encoded Hub call contains neither `newPrice` nor the feeder identity. Once it matures, any feeder already authorized for that pool can choose the price at execution time, and the Hub will read that price from `OracleValuation` while processing the generic revaluation. Managers and sentinels can review the target holding during the delay, but cannot review or veto the actual price that will be used. The authorization therefore does not preserve the manifest's price-control objective, and it is consumed after one use, forcing a new authorization for every update.

Impact

Medium. Any pool using both `StdManifest` and `OracleValuation` loses routine price updates for affected holdings. The resulting stale holding values and accounting state can produce stale NAV and disrupt operations that rely on current asset valuations. Pre-authorizing the generic call restores only a one-shot ability for an existing feeder to supply an arbitrary price, not a timelocked approval of that price.

Recommendation

Add an explicit policy branch for a configured trusted `OracleValuation` address that permits only its `updateHoldingValue()` calls. This makes the feeder mapping the deliberate

authorization boundary for price selection. If price changes must instead be timelocked, refactor the flow so the authorized operation includes the proposed price before it is committed.

Developer Response

Fixed in dc7a853e.

2.7 Low Findings

2.7.1 Permissionless NAV recomputes can refresh the manifest share-price baseline and delay guarded commits

Technical Details

`NAVManager.updateHoldingValue()` is a permissionless function at `src/hooks/accounting/NAVManager.sol:161-163` that forwards directly to `hub.updateHoldingValue()`. When a pool grants `NAVManager` manager rights and uses `StdManifest` in `onchainAccounting` mode, `Hub._protected()` authenticates the immediate caller via `msgSender()`, allowing any external EOA to indirectly exercise the `NAVManager` contract's privileged Hub accounting path.
In the standard on-chain accounting configuration where `NAVManager` is installed as the snapshot hook and a NAV hook such as `SimplePriceManager` is configured, `Hub.updateHoldingValue()` at `src/core/hub/Hub.sol:368-375` unconditionally calls `holdings.callOnSyncSnapshot()` even when the holding difference is zero. If snapshot mode is active for the targeted network, this triggers `NAVManager.onSync()`, which propagates to `SimplePriceManager.onUpdate()` at `src/hooks/accounting/SimplePriceManager.sol:34-60`. That NAV hook always calls `hub.updateSharePrice()` with the recomputed share price, even when the recomputed value equals the already committed price.
`StdManifest.enforce()` at `src/manifests/StdManifest.sol:161-176` treats accounting selectors from `navManager` as always in-policy, so the synchronous execution path reaches `_checkSharePrice()` at `src/manifests/StdManifest.sol:290-312`. After every executed `updateSharePrice()`, the manifest unconditionally refreshes `lastPriceUpdate` to `block.timestamp`, even when the share price did not change. An attacker can therefore send no-op recompute transactions to keep `lastPriceUpdate` artificially fresh.
When a later legitimate share-price update reflects a real valuation change, `_checkSharePrice()` now measures the elapsed time from the attacker-refreshed baseline rather than the last genuine price move. If `priceDelta / elapsed` exceeds `thresholdPerSecond` or `priceDelta` exceeds `maxAbsolutePriceDelta`, the manifest classifies the commit as out-of-policy. Any same-block update is automatically out-of-policy because `elapsed == 0` cannot be rate-bounded. The price commit then requires a matured timelock authorization instead of executing synchronously, delaying the Hub's committed share price.

Impact

Low. Any outsider can delay routine share-price commits on affected pools without holding manager or feeder privileges. By refreshing the manifest's rate-limit baseline with no-op recomputes before legitimate price-sync transactions, an attacker can force otherwise in-policy price updates into the timelocked authorization flow. This keeps committed Hub share prices stale and can temporarily block or delay downstream workflows that depend on timely committed prices.

Recommendation

Prevent no-op recomputes from moving the guard baseline while preserving permissionless recompute behavior if operationally required. The safest fix is to modify `SimplePriceManager.onUpdate()` to skip `hub.updateSharePrice()` when the recomputed share price equals the currently committed price:

```
D18 pricePoolPerShare_ = pricePoolPerShare(poolId);
(D18 currentPrice,) = shareClassManager.pricePoolPerShare(poolId, scId);
if (D18.unwrap(pricePoolPerShare_) != D18.unwrap(currentPrice)) {
    hub.updateSharePrice(poolId, scId, pricePoolPerShare_, uint64(block.timestamp));
}
```

Alternatively, modify `StdManifest` to update `lastPriceUpdate` only when the executed share price actually differs from the prior committed price. If public pokes are not operationally required, additionally gate `NAVManager.updateHoldingValue()` behind a keeper or manager role to restrict the entrypoint.

Developer Response

Fixed in dfe7ca6.

2.7.2 `SimplePriceManager` cannot execute timelocked price exceptions

When on-chain accounting uses a nonzero `StdManifest` share-price guard, an over-limit `SimplePriceManager` update cannot practically use the intended authorization path. The authorization is bound to exact calldata, while the manager fixes `computedAt` to the execution block timestamp.

Technical Details

`StdManifest.enforce()` requires on-chain-accounting price updates to originate from the configured `simplePriceManager`, but `_classify()` still routes those updates through `_checkSharePrice()`. A cap or rate violation returns `delay`, so execution requires a matured authorization.
`HubRegistry.authorize()` and `HubRegistry.consumeAuthorization()` identify an authorization by hashing the complete Hub calldata. The authorized call must therefore include the same `computedAt` value as the later execution.
`SimplePriceManager.onUpdate()` does not accept a caller-selected timestamp. It always calls `Hub.updateSharePrice()` with `uint64(block.timestamp)`. A pool manager scheduling the authorization before its delay elapses cannot generally know the execution block timestamp or the exact NAV-derived price that the later callback will calculate. The eventual SimplePriceManager call consequently has a different authorization id and reverts when it remains outside the configured cap or rate.
With both share-price guards set to zero, the update is correctly instant and this issue does not apply. The affected configuration is on-chain accounting combined with a nonzero guard and an expectation that exceptional price movements can execute after the normal timelock.

Impact

Low. In the affected configuration, an over-limit automatic share-price update behaves as a permanent circuit breaker rather than a timelocked exception. NAV-driven pricing can remain stale until a subsequent update is within the guard or the manifest configuration is changed.

Recommendation

Choose and enforce one model. If over-limit SimplePriceManager updates are intended to be permanently blocked, document that model and require both guards to be zero for continuous automatic pricing. If they are intended to be timelock-executable, use an authorization identity that excludes `computedAt` while remaining bound to the pool, share class, price, and configured SimplePriceManager, or provide a separately authenticated exception path with a deterministic timestamp.

Developer Response

Fixed in 15e5fc86.

2.7.3 `notifyShareClass()` bypasses the timelock for initial share hooks

`StdManifest` classifies `notifyShareClass()` as in-policy even though its manager-supplied `hook` argument becomes the initial transfer hook of the newly deployed spoke share token. A compromised pool manager can therefore install an arbitrary transfer hook without the delay and sentinel veto required for `updateShareHook()`.

Technical Details

`StdManifest._classify()` returns `0` for `IHub.notifyShareClass.selector`.
`Hub.notifyShareClass()` accepts a `bytes32 hook` argument and forwards it unchanged through `sendNotifyShareClass()`.
When the message reaches the spoke, `Spoke.addShareClass()` deploys the share token and calls `shareToken_.file("hook", hook)` whenever the supplied hook is nonzero. The hook participates in ERC-20 transfer checks and can make transfers revert or otherwise impose arbitrary hook behavior.
This is inconsistent with the explicit `IHub.updateShareHook.selector` branch, which returns the manifest delay. The duplicate-registration check in `Spoke.addShareClass()` prevents replacing the hook of an existing token, but does not protect the initial hook. A pool manager can wait for a legitimately authorized `addShareClass()` call to execute and then immediately provision that share class to a spoke with a malicious hook.

Impact

Low. A compromised pool manager can install an arbitrary transfer hook for a new spoke share class without pre-authorization, a delay, or a sentinel veto. The hook can block transfers or impose unintended transfer behavior on the newly deployed token.

Recommendation

Do not classify a manager-selected initial hook as a pure notification. Return the standard delay from `notifyShareClass()` when `hook` is nonzero, or bind the initial hook to separately timelocked pool configuration. If instant token deployment is required, allow only a zero hook through `notifyShareClass()` and install a nonzero hook through the existing supervised `updateShareHook()` path.

Developer Response

Fixed in cccd1dc0.

2.8 Gas Savings Findings

None.

2.9 Informational Findings

2.9.1 Dependency setters can leave `StdManifest` using stale addresses

Technical Details

`StdManifest` stores its `shareClassManager`, `requestManager`, and `bridgingHook` references as immutables. However, `Hub.file("shareClassManager", ...)`, `Hub.setRequestManager`, and `Hub.setBridgingHook` can update the corresponding live dependencies without updating the installed manifest.
Consequently, the manifest may continue reading prices from the old `ShareClassManager` or classifying calls based on obsolete request manager and bridging hook addresses. The setters do not document that rotating these dependencies requires deploying and installing a new manifest. An existing `StdManifestFactory` also retains the old `ShareClassManager`.

Impact

Informational.

Recommendation

Document the coupling between these setters and `StdManifest`. When rotating a pinned dependency, deploy a new factory if necessary and replace the affected manifests. Alternatively, have the manifest retrieve dependencies dynamically from `Hub` or `HubRegistry`.

Developer Response

Documented in a15fc376.

2.9.2 `BridgeCircuitBreaker` does not implement a sliding window rate limiter

Technical Details

`BridgeCircuitBreaker` documents its rate limiter as a rolling window, and `_consumeRateLimit` forwards each transfer to `CircuitBreakerGuard.tally` for accounting.
However, `CircuitBreakerGuard.tally` only stores a single `windowStart` and cumulative `total`. Once `block.timestamp - windowStart > window`, it resets the full tally. This implements a fixed, discrete window anchored to the first transfer in the current period, not a true sliding window over the last `window` seconds.
For example, if two half-limit transfers occur at different times in the same window, the first transfer determines the reset boundary. Immediately after that boundary passes, the full limit becomes available again, even though the later half-limit transfer would still be counted by a true sliding window.

Impact

Informational.
The implementation differs from the documented behavior and may make operators overestimate how smoothly bridge throughput is limited over time.

Recommendation

Update the documentation and comments to describe the current fixed-window behavior, or change the accounting logic to implement a true sliding window if that behavior is required.

Developer Response

Documented in 731c6977.

2.9.3 `StdManifest` does not enforce a longer escalation delay

Technical Details

`StdManifest` documents `escalation` as a longer delay applied when replacing the manifest. However, its constructor assigns `config.delay` and `config.escalation` without validating their relationship.
Consequently, a manifest can be deployed with `escalation <= delay`, allowing manifest replacements to execute no later than ordinary out-of-policy operations and contradicting the documented security model.

Impact

Informational.

Recommendation

Require `config.escalation > config.delay` in the constructor.

Developer Response

Fixed in 267fdc9d.

2.9.4 HubRegistry authorization ids are malleable across equivalent calldata

`HubRegistry` computes authorization ids over the exact calldata bytes supplied to `authorize()`. Since Solidity ABI decoding accepts trailing bytes, a manager can append unused junk to the `data` payload and create multiple distinct pending authorization ids for the same decoded Hub action.

Technical Details

`HubRegistry._authId()` hashes `poolId_.raw()`, the current manifest address, and the raw `data` bytes. `authorize()` rejects only an existing `authorizedAfter[id]`, so uniqueness is enforced over byte-identical payloads rather than over the decoded action.
The Hub and manifest generally decode only the expected arguments for the selected function. For example, a canonical `setManifest()` call and the same call with trailing junk bytes decode to the same selector and arguments, but they produce different authorization ids because `_authId()` includes the trailing bytes. The same property applies to other ABI-decoded Hub calls where extra trailing calldata is ignored by the function body.
This does not allow a different raw payload to consume an existing authorization, because `StdManifest.enforce()` passes the executing `msg.data` to `consumeAuthorization()`. It does allow duplicate pending authorizations for the same semantic action, and a sentinel must cancel each exact raw `data` value emitted in `AuthorizationScheduled`.

Impact

Informational.

Recommendation

If authorization uniqueness is intended to be semantic, canonicalize `data` before computing the id or reject non-canonical calldata. If byte-exact uniqueness is intentional, document that sentinels must cancel the exact raw `data` from each `AuthorizationScheduled` event and that multiple authorizations may coexist for the same action.

Developer Response

Documented in 7de18121.

yAudit

Centrifuge v3.3

Completed 2026-06-10