Security Assessment

Final Report

Silo – Aggregator and Manageable Oracle

February 2026

Prepared for Silo

Table of Contents

Project Summary

Project Scope

Project Name	Repository Link	Commit Hash	Platform
Silo - Aggregator and Manageable Oracle	https://github.com/silo-finance/silo-contracts-v3	PR-1746 (c2bc4b1a61) PR-1748 (d8e16a4cb8)	Solidity

Project Overview

This document describes the security review of **Silo - Aggregator and Manageable Oracle**. The work was undertaken from **February 16th, 2026** to **February 18th, 2026**.

The following contract list is included in our scope:

silo-oracles/contracts/_common/Aggregator.sol
silo-oracles/contracts/chainlinkV3/ChainlinkV3Oracle.sol
silo-oracles/contracts/custom/sAVAX/sAVAXOracle.sol
silo-oracles/contracts/dia/DIAOracle.sol
silo-oracles/contracts/erc4626/ERC4626Oracle.sol
silo-oracles/contracts/erc4626/ERC4626OracleHardcodeQuote.sol
silo-oracles/contracts/erc4626/ERC4626OracleWithUnderlying.sol
silo-oracles/contracts/erc4626/ERC4626OracleWithUnderlying.sol
silo-oracles/contracts/forwarder/OracleForwarder.sol
silo-oracles/contracts/oracleForQA/OracleForQA.sol
silo-oracles/contracts/pendle/linear/PTLinearOracle.sol
silo-oracles/contracts/pendle/lp-tokens/wrappers/PendleWrapperLPTToAssetOracle.sol
silo-oracles/contracts/pendle/lp-tokens/wrappers/PendleWrapperLPTToSyOracle.sol
silo-oracles/contracts/pendle/lp-tokens/PendleLPTOracle.sol

silo-oracles/contracts/manageable/ManageableOracle.sol
silo-oracles/contracts/manageable/ManageableOracleFactory.sol

The team performed a manual audit of all the files in scope**.** During the manual audit, the Certora team discovered bugs in the code, as listed on the following page.

Protocol Overview

The audit focused on two oracle-related changes: the **Aggregator** and the **Manageable Oracle**.

The **Aggregator** introduces a common Chainlink interface that enables the connection of multiple underlying oracles to generate a price.

The **Manageable Oracle** allows the oracle owner to change the underlying oracle. This functionality is intended to rescue markets in cases where the oracle stops operating as expected.

The change of the underlying oracle is subject to a timelock, providing users with time to react before the update takes effect.

Assessment Methodology

Our assessment approach combines design level analysis with a deep review of the implementation to ensure that a protocol is secure, economically sound, and behaves as intended under realistic conditions.

At the design level, we evaluate the architecture, the economic assumptions behind the protocol, and the safety properties that should hold independently of a specific chain or environment. This process includes reviewing internal and cross protocol interactions, state transition flows, trust boundaries, and any mechanism that could be exploited to extract value, deny service, or alter core system behavior. At this stage, a focused threat modelling exercise helps identify key attack surfaces and adversarial capabilities relevant to the system. Design level issues often relate to incentive structures, governance implications, or systemic behavior that emerges under adversarial conditions.

Implementation analysis focuses on the concrete behavior of the code within the execution model of the target chain. This involves reviewing the correctness of logic, access control, state handling, arithmetic behavior, and the nuanced behaviors of the chain environment. Familiar classes of vulnerabilities such as reentrancy conditions, faulty permission checks, precision issues, or unsafe assumptions often surface at this layer. These findings require context aware reasoning that takes into account both the code and the architectural intent.

To support this analysis, the codebase is examined through repeated manual passes and supplemented by automated tools when appropriate. High-risk logic areas receive deeper scrutiny, invariants are validated against both design intent and actual implementation, and potential vulnerability leads are thoroughly investigated. Automated techniques such as static analysis, fuzzing, or symbolic execution may be used to complement manual review and provide additional insight.

Collaboration with the development team plays an important role throughout the audit. This helps confirm expected behaviors, clarify design assumptions, and ensure an accurate understanding of the protocol's intended operation. All findings are documented with clear reasoning, reproducible examples, and actionable recommendations. A follow up review is

conducted to validate the applied fixes and verify that no regressions or secondary issues have been introduced.

Threat and Security Overview

Actors

- **Owner / Admin (trusted role)**: can propose/cancel/accept oracle config changes; can renounce ownership (subject to proposal state).
- **Borrowers**: open/manage leveraged positions; can be adversarial via timing/MEV.
- **Liquidity providers**: supply collateral liquidity (if applicable) and receive share tokens; exposed to accounting/oracle integrity.
- **Liquidators / Keepers**: monitor health and liquidate; can be adversarial, MEV-enabled, or cartelized.

Trust Assumptions

- **The owner is honest and available**: will not censor users, grief with proposals, or intentionally trigger mass liquidations.
- **Oracle correctness and integrity**: prices are not manipulable, stale, or adversarial; validation logic matches oracle behavior.
- **Oracle interface compatibility**: `quote()` (or equivalent) behaves as expected and won't revert/return malformed data under valid conditions.
- **Price decimal convention holds**: `quote()` always returns **18-decimal** prices (no drift across oracle versions/configs).
- **Token standards compliance**: collateral/debt tokens are standard ERC20s and behave predictably (transfer/transferFrom/return values).
- **`decimals()` exists and is reliable**: tokens implement `decimals()` correctly and consistently (no reverts, no spoofing).

Attack Vectors

- **Oracle governance deadlock / liveness failure**
  - **Proposal overwrite blocked**: admin cannot create a new proposal, which will override an old one.
  - **Ownership renounce validation**: Ownership cannot be renounced while there are pending proposals.
  - **Cancel-anytime abuse**: cancelling proposals at any time can be used for **censorship**, uncertainty.
  - **Accept-after-timelock**: Proposal must only be accepted once the timelock has passed.
- **Frontrunning / sandwiching around oracle changes**
  - Users may open loans which will be instantly **insolvent**.
- **Instant liquidation cascades on oracle transition**
  - **Config flip liquidation event**: switching oracle/verification parameters causes abrupt price shifts → mass eligibility and bad debt risk.
- **Oracle behavior/validation failures**
  - **`oracleVerification` bypass**: verification not enforced upon accepting oracle.
  - **Revert/DoS from oracle**: incompatible interface or unexpected revert path halts borrows/liquidations (protocol-wide DoS).
  - **Verification completeness:** All mandatory verification must be performed, preventing a DOS on Silo.
- **Decimal/precision assumptions exploited**
  - **18-decimal price mismatch**: if quote returns non-18 decimals, LTV/health computed wrong → **undercollateralized borrowing** or unfair liquidations.
- **Economic griefing & censorship via admin controls**

- **Indefinite disruption via proposal churn**: repeated propose/cancel cycles create persistent uncertainty and operational instability.
- **Selective timing attacks**: admin times oracle acceptance when specific accounts are vulnerable (targeted liquidations).

- **Integration/compatibility attacks**
  - **Oracle interface mismatch**: wrong return types/units/assumptions across oracle implementations cause silent mispricing.
- **Deployment griefing**
  - **DoS of ManageableOracle creation:** inability to create new ManageableOracle instances via the factory.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	-	-	-
Low	2	2	-
Informational	5	5	-
Total	7	7	-

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High

Detailed Findings

ID	Title	Severity	Status
L-01	beforeQuote() call not validated	Low	Acknowledged
L-02	Changing the oracle may invoke instant liquidations	Low	Acknowledged
I-01	create() may use an underlying oracle with an unexpected address	Informational	Acknowledged
I-02	Same check performed twice in initialization	Informational	Acknowledged
I-03	Unnecessary check in acceptOwnership()	Informational	Acknowledged
I-04	Oracle is only validated during proposal	Informational	Acknowledged
I-05	Oracle cannot be changed during emergency	Informational	Acknowledged

Low Severity Issues

L-01: beforeQuote() call not validated

Severity: Low	Impact: Low	Likelihood: Low
Files: ManageableOracle.sol	Status: Acknowledged	

Description:

The beforeQuote function is invoked in Silo before fetching the oracle price. A reverting beforeQuote() would cause the whole Silo to revert as mentioned in this warning:

```
    ///      WARNING: reverts are propagated to Silo so if `beforeQuote` reverts,
Silo reverts as well.
```

Recommendations:

Consider validating that the beforeQuote function does not revert in oracleVerification().

Customer Response:

Acknowledged.

L-02: Changing the oracle may invoke instant liquidations

Severity: Low	Impact: Low	Likelihood: Low
Files: ManageableOracle.sol	Status: Acknowledged	

Description:

Before changing the oracle, there is a timelock, which allows users to exit their positions. However, some users may not have enough liquidity to repay their positions on time. A change of the oracle may result in an instant price change, which could cause instant liquidations. Furthermore, the oracle update may be sandwiched to create insolvent positions by doing the following flow:

1. User borrows up to the max ltv
2. User withdraws up to the ltv
3. Oracle is updated and now the user is insolvent

Recommendations:

Consider enforcing solutions such as a grace period with disabled liquidations.

Customer Response:

Acknowledged.

Informational Severity Issues

I-01: create() may use an underlying oracle with an unexpected address

Description:

The predictAddress function allows the deployer to predict the address of the underlying oracle that will be deployed by the PTLinearOracleFactory. However, if a user frontruns the ManageableOracle create() function, with a deployment in the PTLinearOracleFactory, which has the same config, the underlying oracle will not be the same as the predicted address due to resolving the existing oracle:

```
 function create(DeploymentConfig memory _deploymentConfig, bytes32
_externalSalt)
        external
        virtual
        returns (IPTLinearOracle oracle)
        ...
 address existingOracle = resolveExistingOracle(id);

        if (existingOracle != address(0)) return IPTLinearOracle(existingOracle);
```

For example:

- Alice calls PTLinearOracleFactory.`predictAddress()` offchain. It predicts the underlying oracle address will equate to `0xALICE`.
- Alice initiates PTLinearOracleFactory.`create()` transaction.
- Bob frontruns and calls PTLinearOracleFactory.`create()` with same `_deploymentConfig`.
- Due to Bob's transaction, the underlying oracle is deployed at `0xBOB`.
- Alice's call completes and returns oracle address at `0xBOB`, causing confusion for Alice.

This affects ManageableOracleFactory because its create() function allows specifying an underlyingOracleFactory and _underlyingOracleInitData. This calls the specified oracle factory via low-level call to deploy/initialize the oracle. Therefore, if Alice expects to deploy the underlying oracle through ManageableOracleFactory.create(), there is a chance that the underlying oracle will be deployed directly by another party and will resolve to a different address than previously expected.

Recommendations:

During the creation of manageable oracles, do not assume the address of the underlying oracle.

Customer Response:

Acknowledged

I-02: Same check performed twice in initialization

Description:

In initialize() the zero oracle validation is performed twice:

```
require(address(_oracle) != address(0), ZeroOracle());
```

And a second time in:

```
function oracleVerification(ISiloOracle _oracle) public view virtual {
        address baseTokenCached = baseToken();
        require(baseTokenCached != address(0), ZeroBaseToken());

        require(address(_oracle) != address(0), ZeroOracle());
```

Recommendations:

Consider removing the first check.

Customer Response:

Acknowledged.

I-03: Unnecessary check in acceptOwnership()

Description:

acceptOwnership() must be called by the new owner:

```
    require(pendingOwnership.value == msg.sender, OnlyOwner());
```

However, the following check serves no value:

```
        require(pendingOwnership.value != address(0),
InvalidOwnershipChangeType());
```

Recommendations:

Consider removing the redundant check.

Customer Response:

Acknowledged.

I-04: Oracle is only validated during proposal

Description:

When proposing an oracle, the following validation is performed:

```
 function oracleVerification(ISiloOracle _oracle) public view virtual {
        address baseTokenCached = baseToken();
        require(baseTokenCached != address(0), ZeroBaseToken());

        require(address(_oracle) != address(0), ZeroOracle());
        require(_oracle.quoteToken() == quoteToken, QuoteTokenMustBeTheSame());
        require(Aggregator(address(_oracle)).baseToken() == baseTokenCached,
BaseTokenMustBeTheSame());

        // sanity check
        try _oracle.quote(10 ** baseTokenDecimals, baseTokenCached) returns
(uint256 price) {
            require(price != 0, OracleQuoteFailed());
        } catch (bytes memory reason) {
            RevertLib.revertBytes(reason, OracleQuoteFailed.selector);
        }
    }
```

However, no validation is performed when the oracle is accepted via the acceptOracle() function. Due to possible mutability of the oracle, this may potentially bypass some of the require statements.

Recommendations:

Consider invoking oracleVerification() in acceptOracle().

Customer Response:

Acknowledged.

I-05: Oracle cannot be changed during emergency

Description:

In the PR description, it is mentioned that the manageable oracle functionality is intended for emergency cases when the previous oracle stops operating as expected. However, the timelock will always enforce a period during which the faulty oracle will have to be used.

Recommendations:

Consider adding an emergency oracle mechanism that delegates to a second trusted oracle that is operating correctly.

Customer Response:

Acknowledged.

Design Recommendations

Below are some recommendations to expand on the functionality and security of the manageable oracle and aggregator implementation. These recommendations will likely add complexity to the codebase and their implementation may not be desired at this point in time.

1. **Addition of emergency fallback oracle in ManageableOracle.**
    a. The current ManageableOracle implementation allows an admin to update the oracle address used by a market in case the oracle has begun to malfunction. The admin must propose a new oracle address and wait for the specified timelock period to pass before finalizing the oracle change. However, it is anticipated that there will be a period of time when the market will not be able to operate while waiting for the timelock to pass. This waiting period can last from 1 to 14 days, depending on the current timelock value.

    Therefore, it is recommended to implement an emergencyOracle storage variable that can be activated by the admin at any time (without a timelock). We recommended that the admin propose the emergency oracle through the same means as proposing a main oracle, requiring the same oracle verification, timelock period, and acceptance methodology. This will impose slight risk to the user, requiring that they perform the same due diligence to check the proposed emergency oracle as they perform for the main oracle.

    b. Expanding on **point a**, it is possible to allow the switch from the main oracle to the emergency oracle occur automatically. For example, if the main oracle reverts for any reason, the emergency oracle can be activated via logic in the contract. For added security, the contract logic could mandate the specific revert conditions that would warrant the switch. Specifically, the logic could check if the revert was due to oracle staleness or sharp price divergence from the previously-recorded value. Regarding potential price divergence, this would require storing historical price data which would add significant gas cost to each call.
    c. Additionally, we considered the idea of enabling preview functionality, displaying the ltv and health of the borrower using the emergency/pending oracles. This will allow

users to timely monitor their positions, preventing instant unexpected liquidations during such changes.

d. We also explored the idea of pausability within Silo. Logically, it would make the most sense to add pause functionality as a hook within the Silo system. This would give the deployer of the silo a way to pause and unpause market operations during times of market uncertainty.

   An alternative approach would be to add pause functionality directly to the ManageableOracle contract since the oracle is called for all borrow and liquidation operations. Allowing the owner to pause the operation of the oracle effectively pauses the borrowing functionality of a Silo. Pausing usually needs to occur as soon as possible in the case of a black swan event, therefore it is not recommended to place the functionality behind a timelock.

   However, it is important to note that pausability within CDP protocols tends to have side effects such as accruing of bad debt due to paused liquidations, as well as instant liquidations which will occur immediately after the protocol is unpaused, without giving users any time to react.

   A rough implementation that would require little code deviation from the above recommendation:

   - Allow setting the emergency oracle to non-zero address behind timelock. This prevents the ability for the owner to rug by setting a malicious oracle immediately.
   - Allow setting the emergency oracle to the zero address immediately.
   - Allow switching from the main oracle to the set emergency oracle immediately.

   In other words, the admin is able to activate the emergency oracle set at address(0) immediately. This would cause immediate reverts following any attempt to borrow or liquidate. The main oracle can be re-activated after market conditions return to normal.

2. Separation of “version” functions in oracles inheriting Aggregator.sol.

The Aggregator contract implements version() that returns a uint256. Meanwhile, all Silo oracle contracts already implement a VERSION() function which returns a string. There is no security issue due to having both functions, however, downstream integrators might be confused about which version function to use. Since the purpose of PR 1746 was to impose the common Chainlink aggregator interface on existing Silo oracles, it makes sense that this one should stay as "version". However, the Silo oracle "VERSION" function might be renamed to siloOracleVersion() or similar to reduce ambiguity.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.