certora

Security Assessment Final Report

Silo

Silo – Kink Model

September-October 2025

Prepared for Silo Team

Table of Contents

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Initial Commit Hash
Silo - Kink Model	[Repo]()	[8e87c7da21]()	[f12498e302]()

Project Overview

This describes the manual code review findings for the **Silo - Kink Model** project. The work was undertaken from **September 29th** to **October 6th.**

The following contract list is included in our scope:

- `contracts/interestRateModel/kink/DynamicKinkModel.sol`
- `contracts/interestRateModel/kink/DynamicKinkModelConfig.sol`
- `contracts/interestRateModel/kink/DynamicKinkModelFactory.sol`
- `contracts/lib/KinkMath.sol`
- `contracts/interfaces/IDynamicKinkModel.sol`
- `contracts/interfaces/IDynamicKinkModelConfig.sol`
- `contracts/interfaces/IDynamicKinkModelFactory.sol`

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

Protocol Overview

The audit focused on evaluating the smart contracts implementing the Dynamic Kink Model, a newly introduced interest rate mechanism designed to adapt to changing market conditions. This model dynamically adjusts the interest rate slope parameter k based on the system's utilization level, increasing or decreasing it depending on whether the utilization falls above or below a defined optimal interval [u1, u2].

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	2	2	1
Low	1	1	0
Informational	5	5	2
Total	8	8	3

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
M-01	New config will be applied retroactively	Medium	Acknowledged
M-02	K is not adjusted when there is no debt	Medium	Fix confirmed.
L-01	verifyConfig() does not validate u1 and u2 against ulow and ucrit	Low	Acknowledged

Audit Goals

1. **Whitepaper correctness**
    a. Verify that the mathematics in the whitepaper are correct, that there are no mathematical mistakes.
2. **Whitepaper corresponds to code**
    a. Ensure the pseudo-code in the whitepaper matches the calculations described in the whitepaper.
        i. Ensure slope (k) only changes when the utilization is outside the optimal interval, and in the correct direction.
    b. Verify the solidity code matches the whitepaper's pseudo-code.
    c. Ensure limits set by the whitepaper are enforced in the code, including both numerical limits and limits between configurable parameters.
3. **Integration correctness**
    a. Ensure the integration with the rest of the Silo protocol is correct.
    b. Ensure the DynamicKinkModel contract correctly implements the IInterestRateModel interface (despite not inheriting from it).
    c. Ensure configuration updates are handled correctly.

Coverage and Conclusions

1. **Whitepaper correctness**
    a. The whitepaper's mathematics are correct. After the fixes in I-04, I-05 the whitepaper is very clear.
2. **Whitepaper corresponds to code**
    a. The whitepaper's pseudocode matches the descriptions given in the whitepaper (after fixing I-05). This includes correctly changing the slope (after fixing the edge case described in M-02).
    b. It has been verified that the solidity code matches the pseudo-code (including all three math-heavy functions, i.e., currentInterestRate, compoundInterestRate, generateConfig).
    c. Limits set by the whitepaper are properly enforced, with the exception of L-01 (some optimal interval limits apply only to user-friendly configs).
3. **Integration correctness**
    a. The new interest rate model correctly integrates with the rest of the Silo protocol. Additional safeguards have been suggested in I-01, I-02 to minimize the potential for configuration mistakes during deployment.
    b. It has been verified that the DynamicKinkModel contract correctly implements the IInterestRateModel interface.
    c. Problems with configuration updates, including possible deployment-time mitigations, have been described in M-01.

Medium Severity Issues

M-01 New config will be applied retroactively		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: DynamicKinkModel.sol	Status: Acknowledged	

Description: When _getCompoundInterestRate() is invoked, it will fetch the ModelState and the config using the following function:

```JavaScript
 function getModelStateAndConfig(bool _usePending)
        public
        view
        virtual
        returns (ModelState memory state, Config memory config, ImmutableConfig memory
immutableConfig)
    {
        ...
            irmConfigToUse = irmConfig();
            state = modelState();
    }
```

Both irmConfig() and modelState() return values relative to the current block timestamp. The configuration (irmConfig) only updates after the activateConfigAt timestamp is reached:

```JavaScript

function irmConfig() public view returns (IDynamicKinkModelConfig config) {
        config = pendingConfigExists() ? configsHistory[_irmConfig].irmConfig : _irmConfig;
    }
```

```
function pendingConfigExists() public view returns (bool) {
        return activateConfigAt > block.timestamp;
    }
```

However, the interest rate computation spans the entire period between _interestRateTimestamp and block.timestamp, which may overlap both the old and new configurations. Despite this overlap, the function applies only the new configuration retroactively across the entire period — effectively ignoring the old configuration for the portion of time before activateConfigAt.

Recommendations: In order to avoid retroactive parameter application, consider computing the interest piecewise across configuration boundaries, applying the appropriate parameters for each interval.

Customer's response: Acknowledged, design choice.

Note: When deploying new configurations, deployers and users should be aware of this. We suggest making sure the state was updated recently (and even within the same timestamp, if possible) before updating to a new configuration.

M-02 k is not adjusted when there is no debt		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: [DynamicKinkModel.sol]()	Status: Fix confirmed.	

Description: If there is no debt, the compoundInterestRate function will return the following:

```
JavaScript
// no debt, no interest, overriding min APR
  if (_tba == 0) return (0, _state.k);
```

However, there could be extreme cases in which the debt was decreased instantly after k had reached the kmax value. As a result, the maximum kink will still be preserved even though the utilization is 0.

Recommendations: Consider adjusting k when the utilization is 0, either according to the formula or directly reset to kmin (as this is an extreme edge case).

Customer's response: Fixed in PR-1664

Fix Review: Fix confirmed.

Low Severity Issues

L-01 verifyConfig() does not validate u1 and u2 against ulow and ucrit		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: DynamicKinkModel.sol	Status: Acknowledged	

Description: generateConfig() performs the following validations:

```
JavaScript
// 0 <= ulow < u1 < u2 < ucrit < DP
    require(defaultInt.u1.inOpenInterval(defaultInt.ulow, defaultInt.u2),
IDynamicKinkModel.InvalidU1());
    require(defaultInt.u2.inOpenInterval(defaultInt.u1, defaultInt.ucrit),
IDynamicKinkModel.InvalidU2());
    require(defaultInt.ucrit.inOpenInterval(defaultInt.u2, DP),
IDynamicKinkModel.InvalidUcrit());
```

However, in verifyConfig() there are no checks to ensure that u1 and u2 are between ulow and ucrit:

```
JavaScript
require(_config.ulow.inClosedInterval(0, _DP), InvalidUlow());
require(_config.u1.inClosedInterval(0, _DP), InvalidU1());
require(_config.u2.inClosedInterval(_config.u1, _DP), InvalidU2());
require(_config.ucrit.inClosedInterval(_config.ulow, _DP), InvalidUcrit());
```

As a result, an incorrect configuration can be created.

Recommendations: Consider adding the additional validations into verifyConfig(), in order to ensure that u1 and u2 are between ulow and ucrit.

Customer's response: Acknowledged - design choice to allow more flexibility for the configurations.

Informational Issues

I-01. _create() does not verify the silo address and the immutableArgs

Description: _create() will perform some initial validations to ensure that initialization will not revert:

```javascript
function _create(
        IDynamicKinkModel.Config memory _config,
        IDynamicKinkModel.ImmutableArgs memory _immutableArgs,
        address _initialOwner,
        address _silo,
        bytes32 _externalSalt
    ) internal virtual returns (IDynamicKinkModel irm) {
        IRM.verifyConfig(_config);
```

However, it will not perform the following checks, which are done in the initialize function:

```javascript
require(_silo != address(0), EmptySilo());
require(_immutableArgs.timelock <= MAX_TIMELOCK, InvalidTimelock());
require(_immutableArgs.rcompCap > 0, InvalidRcompCap());
require(_immutableArgs.rcompCap <= RCUR_CAP, InvalidRcompCap());
```

As a result, the _create function will not revert early as intended.

Recommendation: Consider adding the additional validations in the beginning of _create.

Customer's response: Acknowledged, design choice.

I-02. Invalid DynamicKinkModel contracts can be deployed by the factory

Description: _getCompoundInterestRate() and _getCurrentInterestRate() will both perform the following external call:

```javascript
ISilo.UtilizationData memory data = ISilo(_silo).utilizationData();
```

However, DynamicKinkModel::initialize() and the factory do not check if the specified _silo address will successfully execute this function. As a result, the factory may deploy broken DynamicKinkModel contracts.

Recommendation: Consider checking if the utilizationData() call is successful during initialization or in _create().

Customer's response: Acknowledged — this is a design limitation. The Silo contract is created (cloned) after the Interest Rate Model (IRM) is deployed, so when the IRM is initialized with the Silo's address, the Silo does not yet exist.

I-03. Interest can be underestimated during long periods of inactivity

Description: The DynamicKinkModel compounds interest continuously on a per-second basis, which incrementally increases the borrowed amount. Consequently, utilization naturally rises over time as compounding progresses. However, the current implementation assumes that utilization remains constant between _t0 and _t1, leading to a slight underestimation of the compounded interest. Although this deviation is generally minor, it can become more significant depending on the configuration parameters and the duration of inactivity.

Recommendation: Consider either documenting this behavior to acknowledge the expected underestimation (currently the utilization rate remaining constant for the calculation is documented as an assumption, and not as an underestimation), or introducing a mechanism to periodically update or re-evaluate the interest rate during periods of low user activity.

Customer's response: Acknowledged. This is a known problem in lending protocols, and will not be fixed.

I-04. Whitepaper accuracy can be improved

Description: According to the whitepaper:
"If utilization is above ucrit, the slope changes from k to αk, where α ⩾ 0 is a high factor."
However, in the implementation, the actual slope applied is *(1 + α) × k*, aligning with the formal definition provided in the static model.

Recommendation: Consider updating the whitepaper to describe the formula more accurately.

Customer's response: Fixed in PR-1675.

Fix Review: Fix confirmed.

I-05. Whitepaper inconsistency

Description: In the "How do we calculate the compound interest in practice?" section, the integral calculations are shown using a r, which is the interest per second, while in the code the slope state is calculated using the kink and later multiplying by the utilization factor:

```JavaScript
if (_l.k1 > _cfg.kmax) {
            _l.x = _cfg.kmax * _l.T - (_cfg.kmax - k) ** 2 / (2 * _l.roc);
            k = _cfg.kmax;
        } else if (_l.k1 < _cfg.kmin) {
            _l.x = _cfg.kmin * _l.T - (k - _cfg.kmin) ** 2 / (2 * _l.roc);
            k = _cfg.kmin;
        } else {
            _l.x = (k + _l.k1) * _l.T / 2;
            k = _l.k1;
        }
if (_u >= _cfg.ulow) {
            _l.f = _u - _cfg.ulow;

            if (_u >= _cfg.ucrit) {
                _l.f = _l.f + _cfg.alpha * (_u - _cfg.ucrit) / _DP;
            }
        }
        _l.x = _cfg.rmin * _l.T + _l.f * _l.x / _DP;
```

Furthermore, due to the use of a negative roc, the following code will perform a subtraction, instead of addition which was mentioned in the whitepaper:

```JavaScript

        } else if (_l.k1 < _cfg.kmin) {
            _l.x = _cfg.kmin * _l.T - (k - _cfg.kmin) ** 2 / (2 * _l.roc);
            k = _cfg.kmin;
```

In addition, the whitepaper could be made clearer and more consistent by explicitly defining all variables (e.g., k_lin, k_i) and by clearly stating that the presented derivation refers to the decreasing slope case. It would also be beneficial to provide a separate explanation for the increasing slope case.

Recommendation: Consider including detailed NatSpec documentation explaining how the code translates the theoretical equations from the whitepaper and improving the whitepaper.

Customer's response: Fixed in PR-1675.

Fix Review: Fix confirmed.

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.