ENIGMA DARK

Securing the Shadows

Best-Efforts Security Review

Silo v2 Core

April 16, 2025

Contents

Summary

Enigma Dark
Enigma Dark is a web3 security firm leveraging the best talent in the space to secure all kinds of blockchain protocols and decentralized apps. Our team comprises experts who have honed their skills at some of the best auditing companies in the industry. With a proven track record as highly skilled white-hats, they bring a wealth of experience and a deep understanding of the technology and the ecosystem.

Learn more about us at enigmadark.com

Silo v2 Core
Silo V2 is a risk-isolated lending protocol designed to scale securely and permissionlessly. Each lending market consists of two immutable Silos, one for each asset, ensuring isolated risk and capital efficiency. Built for composability, Silo V2 enables anyone to launch lending markets through the Silo Factory, providing a modular, permissionless, and secure foundation for DeFi borrowing and lending.

Engagement Overview

Over the course of 1.5 weeks, beginning April 16 2025, the Enigma Dark team conducted a comprehensive, best-effort final security review of the Silo v2 Core project.

This review focused on validating recent fixes and updates stemming from previous audits and internal assessments, while also performing an in-depth pass over critical components of the core codebase. The review was performed by two Lead Security Researchers: vnmrtz & 0xWeiss, and one Security Researcher: kiki.

The following repositories were reviewed at the specified commits:

Repository	Commit
silo-contracts-v2/silo-core	`c6eb7cc7cb8f5ed78f3cbe36a4a54558b32e3b34`

Risk Classification

Severity	Description
Critical	Vulnerabilities that lead to a loss of a significant portion of funds of the system.
High	Exploitable, causing loss or manipulation of assets or data.
Medium	Risk of future exploits that may or may not impact the smart contract execution.
Low	Minor code errors that may or may not impact the smart contract execution.
Informational	Non-critical observations or suggestions for improving code quality, readability, or best practices.

Vulnerability Summary

Severity	Count	Fixed	Acknowledged
Critical	0	0	0
High	0	0	0
Medium	1	1	0
Low	5	2	3
Informational	0	0	0

Findings

Index	Issue Title	Status
M-01	`maxWithdraw` ignores fractional interest, potentially causing DoS on full withdrawals	Fixed
L-01	`maxBorrow` does not account for fractional interest, potentially causing DoS when borrowing the maximum amount	Fixed
L-02	Incorrect requirement when creating and updating programs	Acknowledged
L-03	Accruing rewards could run out of gas	Acknowledged
L-04	Incorrect Liquidation Share Rounding	Acknowledged
L-05	Dao Fee Revenue Can Be Redirected to Deployer	Fixed

Detailed Findings

High Risk

No issues found.

Medium Risk

M-01 - `maxWithdraw` ignores fractional interest, potentially causing DoS on full withdrawals

Severity: Medium Risk

Context:

- Silo.sol#L228

Technical Details:
According to the invariant `LENDING_INVARIANT_B: Result of maxWithdraw() used as input to withdraw() should never revert`, any call to `withdraw` using the return value of `maxWithdraw` should always succeed. This ensures protocol availability for integrators and users aiming to perform maximum withdraw operations, for example in a rebalance kind of logic.

However, due to the `maxWithdraw` view function not accounting for the recently introduced fractions logic, certain edge cases may cause the `withdraw` call to revert.

A proof-of-concept test reproducing this issue is available here.

Impact:
In certain protocol states, users and integrators attempting to withdraw the maximum allowable amount might encounter reverts, reducing reliability and usability of the protocol under those conditions, especially in lending protocols where withdrawals and repayments must remain consistently available.

Recommendation:
Adjust the `maxWithdraw` view function to account for fractions logic. If incorporating full support introduces excessive complexity, a more pragmatic solution would be to underestimate the return value slightly. This ensures that calls relying on `maxWithdraw` remain valid and protocol availability is preserved.

Developer Response:
Fixed at commits `983caba`. Underestimating `maxWithdraw` to count for interest fractions.

After commit `983caba` the invariant suite has been run again to confirm the issue no longer exists and `LENDING_INVARIANT_B` holds.

Low Risk

L-01 - `maxBorrow` does not account for fractional interest, potentially causing DoS when borrowing the maximum amount

Severity: Low Risk

Context:

- Silo.sol#L410

Technical Details:
According to the invariant `BORROWING_HSPOST_F: User borrowing maxBorrow should never revert`, any call to `borrow` using the return value of `maxBorrow` should always succeed. This ensures protocol availability for integrators and users aiming to perform maximum borrow operations.

However, due to the `maxBorrow` view function not accounting for the recently introduced fractions logic, certain edge cases may cause the `borrow` call to revert. Specifically, when calculating fractions, it's possible for totalDebtAssets to increase by 1. This, in turn, can lead to a failure in the Loan-to-Value (LTV) check.

A proof-of-concept test reproducing this issue is available here.

Impact:
In certain protocol states, users and integrators attempting to borrow the maximum allowable amount might encounter reverts, reducing reliability and usability of the protocol under those conditions.

Recommendation:
Adjust the maxBorrow view function to account for fractions logic. If incorporating full support introduces excessive complexity, a more pragmatic solution would be to underestimate the return value slightly. This ensures that calls relying on maxBorrow remain valid and protocol availability is preserved.

Developer Response:
Fixed at commits `15f345f`, `3221567`. To avoid this, we underestimate assets. Having fewer shares is acceptable because it underestimates the value due to missing fractions. When recalculating assets (due to the issue described above), we want a lower share price (which occurs when there are no fractions), as this leads to fewer assets and keeps us within the LTV limit.

Therefore, we decrement `_totalDebtAssets` with `_totalDebtAssets--` offset the earlier `_totalDebtAssets++`.

After commit `3221567` the invariant suite has been run again to confirm the issue no longer exists.

L-02 - Incorrect requirement when creating and updating programs

Severity: Low Risk

Context:

- BaseIncentivesController.sol#L46

Technical Details:

The `createIncentivesProgram` and the `updateIncentivesProgram` functions allow to change the emissions per second, which they are capped at `MAX_EMISSION_PER_SECOND`.

The problem is that in both instances the `emissionPerSecond` is required to be smaller than `MAX_EMISSION_PER_SECOND` while it should be smaller or equal. Normally, when using bounds such as maximum values for uints the max value should be included in the acceptable range for that uint:

```
  function
createIncentivesProgram(DistributionTypes.IncentivesProgramCreationInput
memory _incentivesProgramInput)
        external
        virtual
        onlyOwner
    {
        require(bytes(_incentivesProgramInput.name).length <= 32,
TooLongProgramName());
        require(_incentivesProgramInput.emissionPerSecond <
MAX_EMISSION_PER_SECOND, EmissionPerSecondTooHigh());
        require(_incentivesProgramInput.distributionEnd >= block.timestamp,
InvalidDistributionEnd());
```

Impact:

One off error.

Recommendation:

Update the following code:

```
 require(bytes(_incentivesProgramInput.name).length <= 32,
TooLongProgramName());
-        require(_incentivesProgramInput.emissionPerSecond <
MAX_EMISSION_PER_SECOND, EmissionPerSecondTooHigh());
+       require(_incentivesProgramInput._emissionPerSecond <=
MAX_EMISSION_PER_SECOND, EmissionPerSecondTooHigh());
        require(_incentivesProgramInput.distributionEnd >= block.timestamp,
InvalidDistributionEnd());
```

```
        require(_distributionEnd >= block.timestamp,
InvalidDistributionEnd());
+      require(_emissionPerSecond <= MAX_EMISSION_PER_SECOND,
EmissionPerSecondTooHigh());
-        require(_emissionPerSecond < MAX_EMISSION_PER_SECOND,
EmissionPerSecondTooHigh());
```

Developer Response: Acknowledged.

L-03 - Accruing rewards could run out of gas

Severity: Low Risk

Context:

- BaseIncentivesController.sol#L46

Technical Details:

When creating an incentive program, the program is added to the `_incentivesProgramIds` set, which is basically an array of all the programs that exist:

```
 function _createIncentiveProgram(
        bytes32 _programId,
        DistributionTypes.IncentivesProgramCreationInput memory
_incentivesProgramInput
    ) internal virtual {
        require(_incentivesProgramInput.rewardToken != address(0),
InvalidRewardToken());
        require(_incentivesProgramIds.add(_programId),
IncentivesProgramAlreadyExists());
```

When accruing rewards from a certain user it does loops through all the existing programs:

```
 function _accrueRewards(address _user)
        internal
        virtual
        returns (AccruedRewards[] memory accruedRewards)
    {
        accruedRewards = _accrueRewardsForPrograms(_user,
_incentivesProgramIds.values());
    }
```

In case there is a high enough number of programs, the accrual function could run out of gas not allowing for the `_accrueRewards` function to work.

Impact:

DOS.

Recommendation:

Add a maximum amount of programs check inside the `_createIncentiveProgram` function.

Developer Response: Acknowledged.

L-04 - Incorrect Liquidation Share Rounding

Severity: Low Risk

Context:

- PartialLiquidation.sol#L217

Technical Details:
In the `_callShareTokenForwardTransferNoChecks` function of the `PartialLiquidation` contract, the share calculation rounds the shares to liquidate in the wrong direction. When converting the asset amount to shares, the rounding is specified to round towards the floor — in other words, down.

Because of this, the borrower that is getting liquidated is consistently at an advantage because they end up transferring a value of shares that are less than the value of assets that the liquidator was supposed to receive.

Impact:
Liquidator receives fewer funds for liquidating than expected.

Recommendation:
Consider using `Math.Rounding.Ceil` for `LIQUIDATE_TO_SHARES` and be sure to compare and take the smaller of the rounded-up value and the borrower's total shares. This way,

the borrower is never expected to transfer more shares than they possess.

Developer Response:
Acknowledged. We've decided to save the borrower, as the liquidator still collects the fee.

L-05 - Dao Fee Revenue Can Be Redirected to Deployer

Severity: Low Risk

Context:

- Actions.sol#L424-L480

Technical Details:
In the `withdrawFees` function of the Actions library, the fee distribution mechanism contains a rounding vulnerability that allows a malicious deployer to significantly reduce or eliminate the DAO's share of protocol fees through manipulation of the available liquidity.

The current implementation calculates the DAO's fee portion first, applying division that results in rounding down. The deployer's portion is then derived as the remaining fees. This approach creates a scenario where fee division inherently prioritizes the deployer over the DAO.

This allows for any deployer of a silo with low decimals such as Gemini's GUSD to call the 'withdrawFees' while the available liquidity is low and cause the DAO's portion of the fees to round down to zero, and the total amount of `daoAndDeployerRevenue` gets sent to the deployer instead. The deployer can also leverage flash loans to further manipulate the silo's available liquidity to consistently force the DAO's portion to round down.

Impact:
Protocol DAO revenue is redirected to the deployer.

Recommendation:
Consider prioritizing the DAO portion by either performing the division on the deployer alternatively you can explicitly round the DAO portion up when performing the division.

Developer Response:
Fixed at commit `a0124d2`.

Disclaimer

This report does not endorse or critique any specific project or team. It does not assess the economic value or viability of any product or asset developed by parties engaging Enigma Dark for security assessments. We do not provide warranties regarding the bug-free nature of analyzed technology or make judgments on its business model, proprietors, or legal compliance.

This report is not intended for investment decisions or project participation guidance. Enigma Dark aims to improve code quality and mitigate risks associated with blockchain technology and cryptographic tokens through rigorous assessments.

Blockchain technology and cryptographic assets inherently involve significant risks. Each entity is responsible for conducting their own due diligence and maintaining security measures. Our assessments aim to reduce vulnerabilities but do not guarantee the security or functionality of the technologies analyzed.

This security engagement does not guarantee against a hack. It is a review of the codebase during a specific period of time. Enigma Dark makes no warranties regarding the security of the code and does not warrant that the code is free from defects. By deploying or using the code, the project and users of the contracts agree to use the code at their own risk. Any modifications to the code will require a new security review.