PUBLIC

Code Assessment

of the CurveLend Operators Smart Contracts

Oct 15, 2025

Produced for

RESUPPLY

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Resupply team,

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

Resupply implements Curve Lending Operators and a Curve Lending Operator Factory. The contracts provide infrastructure for Curve to mint crvUSD through existing crvUSD lending markets, by supplying them with unbacked crvUSD, instead of deploying new crvUSD minting markets.

The most critical subjects covered in our audit are correct accounting, rounding behavior, and integration with ERC4626 vaults. Security regarding the aforementioned topics is high. In the contracts, the rounding behavior of deposits and withdrawals is generally ignored, which is benign if the value of shares is not inflated, see notes Deposits Incur Rounding Losses and Rounding Errors in Vault Operations May Cause Withdrawal Failures. A note about a potential pitfall in the integration with Curve DAO Governance is that Deposit Can Revert Because Of Market Supply Limit.

The general subjects covered are events, documentation, naming conventions. Some informational issues have been raised to help improve the aforementioned subject.

In summary, we find that the codebase provides a high level of security.

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

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	0

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 CurveLend Operators repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	14 September 2025	968bb07dc2e17fa30c7687c162460f872151fa2a	Initial Version
2	9 October 2025	dc681dcbad909d33e0f9f698c6290ba2c478ecbd	Fixes

For the solidity smart contracts, the compiler version `0.8.28` was chosen.

The following files are in scope:

```
src/dao/CurveLendMinterFactory.sol
src/dao/CurveLendOperator.sol
```

2.1.1 Excluded from scope

Any contracts that not not explicitly listed in the scope section are excluded from the the scope of this review. All other files in the repository, including other smart contract files and tests, are explicity excluded from the scope of this review.

Further, while the integration with the Curve Lending, crvUSD and the Controller Factory is in scope of this review, the actual code of these external components is out of scope.

In this report, we assume contracts are deployed to Ethereum Mainnet. The correctness of the codebase if deployed on other chains is not in scope of this review.

2.2 System Overview

This system overview describes the latest received version (Version 2) 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.

Resupply offers a `CurveLendMinterFactory` and `CurveLendOperator`, which together manage the process of receiving newly minted `crvUSD`, supplying them to lending markets, and harvesting profits. The factory creates and oversees individual `CurveLendOperators`, each dedicated to a single lending market.

2.2.1 CurveLendMinterFactory

`CurveLendMinterFactory` is an ownable contract that acts as the central hub for managing lending operations. It is responsible for creating `CurveLendOperator` instances and distributing borrowed funds to them.

The key functionalities of the factory are:

- The owner can deploy new `CurveLendOperator` clones (minimal proxies) for specified lending `markets` (Curve Lending) via the `addMarketOperator()` function.
- The factory receives `crvUSD` directly from the `crvusdControllerFactory`. It then allows the operator contracts it manages to `borrow()` these funds. `borrow()` simply transfers the funds to the operator.
- The owner can call `setImplementation()` to update the address of the implementation for the new `CurveLendOperator` clones, and can call `setFeeReceiver()` to designate a `fee_receiver` that receives the profits.
- The owner has the authority to `removeMarketOperator()` or to overwrite an existing `CurveLendOperator` with a new one associated with the same `market`, which prevents the original operator from borrowing additional funds.

2.2.2 CurveLendOperator

The `CurveLendOperator` contract is responsible for managing the funds within a single lending `market`. This includes depositing and withdrawing funds to and from the `market` to match the `mintLimit` set by the operator owner, as well as withdrawing profits. Each operator has the same owner as the `CurveLendMinterFactory`. Each instance is created by the `CurveLendMinterFactory`.

Its primary functions include:

- Depositing `mintLimit` `crvUSD` into its assigned `market`, which is a ERC4626 compliant Curve Lending vault.
- When the owner increases an operator's `mintLimit` using `setMintLimit()`, it automatically borrows the `crvUSD` amount corresponding to the increase of debt limit from the `CurveLendMinterFactory` using `borrow()` and deposits them into the vault.
- After the `mintLimit` has been lowered using `setMintLimit()`, the `reduceAmount()` function can be called by anyone to withdraw any amount of funds from the vault and return them to the factory as long as the total `mintedAmount` exceeds the `mintLimit`.
- The `withdraw_profit()` function can be called by anyone and allows for the withdrawal of any assets that exceed the total `mintedAmount`, sending the profit to the `CurveLendMinterFactory`'s designated `fee_receiver`.

2.3 Trust Model

The owner of the `CurveLendMinterFactory` is **fully trusted**. It has the power to freely move `crvUSD` out of the contract. On Ethereum Mainnet, it is expected to be the Curve Ownership Agent, at address:

- 0x40907540d8a6C65c637785e8f8B742ae6b0b9968

On other chains, owner and fee receiver are expected to be contracts or wallets controlled by Curve DAO.

The markets (vaults) to which `crvUSD` is deposited are expected to be instances of Curve Lending Vaults. They are trusted to be deployed with legitimate parameters. Manipulation of the price oracle underlying the market for the under-collateralized borrowing of assets should be economically unprofitable.

On Ethereum Mainnet, the `crvusdControllerFactory` of `CurveLendMinterFactory` is expected to be the crvUSD ControllerFactory. On other chains it is expected to be a contract or wallet controlled by Curve DAO. It can pull funds from the `CurveLendMinterFactory` because it has unlimited approval.

3 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.

4 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.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Share Inflation Attack via Rounding up in reduceAmount() Code Corrected

Low-Severity Findings	0
Informational Findings	2

- Silent Reduction of Withdrawn Amount Code Corrected
- Typo In Documentation Code Corrected

6.1 Share Inflation Attack via Rounding up in reduceAmount()

Security Medium Version 1 Code Corrected

CS-RESCRV-001

In the `CurveLendOperator` contract, the `reduceAmount()` function allows withdrawals as small as 1 wei from the `market` using `withdraw()`. This creates a vulnerability when combining the rounding behavior in assets to share conversions with a share inflation attacks on the `market`.

This attack needs to be run on a freshly created `market` which has not been seeded yet. The cost would otherwise be too high.

In the Curve Vault implementation, the `_convert_to_shares()` function rounds up when calculating shares to burn during withdrawals.

When withdrawing 1 wei of assets, if the value of 1 share is more than 1 crvUSD, the vault rounds up and burns 1 wei of share, effectively transferring value from the CurveLendOperator to remaining shareholders. Assets are converted to shares using this formula, with `is_floor` set to true in `deposit()` and to false in `withdraw()`, `DEAD_SHARES = 1000` and `precision = 1` in the case of crvUSD:

```
numerator: uint256 = (self.totalSupply + DEAD_SHARES) * assets * precision
denominator: uint256 = (total_assets * precision + 1)
if is_floor:
    return numerator / denominator
else:
    return (numerator + denominator - 1) / denominator
```

Assuming that the `market` was freshly created, an attacker can first inflate the `market`'s price per share to an extreme value (e.g., $5 \cdot 10^{19}$ per share, that is 50 crvUSD per share), then repeatedly call `reduceAmount(1)` to withdraw 1 wei at a time. Due to the vault's rounding-up behavior, each 1-wei withdrawal burns exactly 1 share from the operator and is a loss of $5 \cdot 10^{19}$ for the operator. By

repeating this process, the attacker can burn all operator shares and appropriate part of the operator's deposited funds.

The attack could proceed as follows, with the `market` being a fresh Curve Vault, all values are in crvUSD or share "weis", that is not scaled by 10^{-18}, unless specified otherwise:

1. Attacker deposits the minimum amount, 10 000, in the vault (no other deposits yet). It receives 10 000 000 shares in return. Assets are thus 10 000 and, including the Vault 1000 *dead shares*, 10 001 000 shares are issued.
2. Attacker donates 50 000 000 wei of crvUSD to the vault and then withdraws the same amount to burn most of their shares. Assets are thus back to 10 000 but, including the dead shares, 1999 shares are now issued.
3. Attacker donates 10^{23} to the vault (100k crvUSD) to raise the price per share to $5 \cdot 10^{19}$ (50 crvUSD per wei of share).
4. Operator deposits funds. We assume it receives a correct amount of shares. e.g. if the CurveLendOperator deposits 300K crvUSD ($3 \cdot 10^{23}$), it receives 3000 shares.
5. Admin reduces `mintLimit` below `mintedAmount`, enabling `reduceAmount()` calls.
6. Attacker calls `reduceAmount(1)` repeatedly, until all shares are burned, operator gets 1 wei of crvUSD for each burned share, while the shares were worth 50 crvUSD ($5 \cdot 10^{19}$) each.
7. Attacker redeems their shares for vault assets.

The operator loses 100% of deposited funds while the attacker's withdraws around 50% of the lost funds. Attacker's cost is equal to the amount donated to the dead shares at step 3 (50 crvUSD per dead share, or 50k crvUSD in total), plus gas fees. The 50% parameter is improvable by the attacker by keeping a bigger share of the Vault at step (2), but requires donating more capital (which will then be recovered) at step (3) to achieve the same share price.

If other users deposit funds to the vault during the process, the profit of the attacker is dependent on the percentage of shares owned by other users (who would also profit from the attack). The cost of the attacker is also dependent on the gas fees for the repeated `reduceAmount()` calls: if gas fees are low, the attacker can donate a smaller amount and inflate the price of each share to a lower degree (e.g. donating 10^{21} (1000 crvUSD), resulting in a price per share of $5 \cdot 10^{17}$ (0.5 crvUSD)).

Code corrected:

A sanity check has been addded that ensures significant share inflation has not taken place, and is not possible in the future. The sanity check consists in checking that at least 1000e18 shares, corresponding to approximately 1e18 crvUSD for a non-inflated vault, have been burned to address `0xdead`.

6.2 Silent Reduction of Withdrawn Amount

Informational **Version 1** **Code Corrected**

CS-RESCRV-010

The function `CurveLendOperator.reduceAmount()` silently caps the withdrawal amounts to the available minted balance, contrary to the NatSpec documentation, which describes the parameter `_amount` as the “amount to reduce.”

Since the function is permissionless, a user could frontrun the call to reduce the available minted balance, causing integrators to withdraw less than expected and potentially breaking downstream logic.

Integrators should be expected to snapshot the minted balance before and after the call to determine the amount withdrawn.

Code corrected:

The `reduceAmount()` function returns the amount by which `mintedAmount` was actually reduced.

6.3 Typo In Documentation

Informational | Version 1 | Code Corrected

CS-RESCRV-011

In line 75 of function `_setMintLimit()` of `CurveLendOperator`, "adaquate" should be changed to "adequate".

Corrected:

The typo has been corrected.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Events Lack Operator Information

Informational | Version 1 | Code Partially Corrected

CS-RESCRV-004

In `CurveLendMinterFactory`, events `Borrow` and `RemoveMarket` are missing the address of the concerned `CurveLendOperator`.

Code partially corrected:

A field called `_operator` has been added to the `Borrow` event.

7.2 Gas Optimization

Informational | Version 1 | Acknowledged

CS-RESCRV-005

In function `_setMintLimit()` of `CurveLendOperator`, `mintedAmount += difference` can be changed to `mintedAmount = _newLimit` to save one SLOAD.

Acknowledged:

Resupply acknowledges the potential optimization but choses to not change the code.

7.3 Misleading Variable/Contract Names

Informational | Version 1 | Code Partially Corrected

CS-RESCRV-006

- The contract name `CurveLendMinterFactory` is unexpected since it is the factory contract of `CurveLendOperator`.
- The parameter `_lender` of event `CurveLendMinterFactory.AddMarket` is the only occurence of the term lender in both contracts. It is always referred to as market operator.
- The state variable `crvusdController` in `CurveLendMinterFactory` stores the address of Curve's `ControllerFactory` and not a `Controller` contract.
- The name of variable `currentAssets` in `CurveLendOperator.withdraw_profit()` is misleading, as it first refers to the assets of the operator in the market and later refers to the *profit* extracted.

Partially corrected:

The `_lender` parameter has been renamed to `_operator` and `crvusdController` has been renamed to `crvusdControllerFactory`.

7.4 Missing Sanity Checks

Informational **Version 1** **Acknowledged**

CS-RESCRV-007

1. In the `initialize()` function of `CurveLendOperator`, the following assertion prevents the function from being called more than once:

```
require(market == address(0),"!init");
```

If the `CurveLendOperator` were to be initialized with `market = address(0)`, anyone would be able to call initialize again and gain ownership of the `CurveLendOperator`.

Note that the `CurveLendMinterFactory` enforces that `_market != address(0)` in `addMarketOperator()`. Therefore, as long as the contract is initialized through the factory, it can never be re-initialized.

2. The initializer of the implementation contract of `CurveLendOperator` is not disabled in the constructor. This allows anybody to "initialize" the implementation contract with arbitrary values. However, this has no consequences as the implementation contract has no privileges.
3. The `market` set in `CurveLendOperator` is not validated to have crvUSD as base asset during initialization.

**Acknowledged:*

Resupply acknowledges the issue and chooses to leave the code unchanged.

7.5 RemoveMarket Not Emitted If Operator Is Overwritten

Informational **Version 1** **Acknowledged**

CS-RESCRV-008

In `CurveLendMinterFactory`, the event `RemoveMarket` is emitted when calling `removeMarketOperator()` but is not emitted when calling `addMarketOperator()` with a market already used by another `CurveLendOperator`. This leads to the `CurveLendOperator` being overwritten in `markets`, having effectively the same effect as calling `removeMarketOperator()` and then `addMarketOperator()`, but with no `RemoveMarket` event emitted.

Acknowledged:

Resupply acknowledges the issue and choses to leave the code unchanged.

7.6 Rounding Errors in Vault Operations May Cause Withdrawal Failures

Informational | Version 1 | Acknowledged

CS-RESCRV-009

The crvUSD amount sent to the system under review is tracked in three locations: Curve's ControllerFactory `debt_ceiling_residual` mapping (total crvUSD minted to the `CurveLendMinterFactory`), each CurveLendOperator's `mintedAmount` state variable (crvUSD borrowed by that operator), and each CurveLendOperator's vault balance (crvUSD deposited plus interest earned).

Since borrowing from the CurveLendMinterFactory involves no rounding, in the absence of donations, the sum of all operators' `mintedAmount` plus the minter factory's residual balance equals the factory's total `debt_ceiling_residual`.

However, vault operations introduce small rounding losses where `deposit()` rounds down shares minted and `withdraw()` rounds up shares burned. These cumulative losses cause the crvUSD value of the vault shares of the operator to become slightly less than `mintedAmount`, meaning full withdrawal of the recorded `mintedAmount` may fail.

Since profit is calculated as `vault_balance - mintedAmount`, rounding losses directly reduce reported profits, so profits otherwise paid to the fee receiver will absorb these losses, but in the edge case with no profits (or all profits withdrawn), withdrawal attempts for the full `mintedAmount` may fail due to insufficient vault balance.

Resupply acknowledges the issue and choses to leave the code unchanged.

7.7 Withdraw_Profit Can Be Performed Before reduceAmount

Informational | Version 1 | Acknowledged

CS-RESCRV-003

In case of low liquidity in the vaults, or if bad debt has been incurred, it can become challenging to `withdraw()` assets from the vault. In that case, there is a race condition between debt repayment to the minter (`reduceAmount()`) and taking profit through `withdraw_profit()`.

Acknowledged:

Resupply acknowledges the issue and choses not to modify the code.

8 Notes

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

8.1 Curve Lending Bad Debt Is Not Socialized

Note Version 1

CurveLendOperators "mint" crvUSD to Curve Lending markets, as opposed to Curve Minting markets. The lending markets might have riskier parameters, and be more likely to generate bad debt. In case bad debt is generated in Curve Lending markets, it is not socialized, and "bank runs" can leave the last depositors unable to withdraw. Since CurveLendOperators depend on governance actions to withdraw from Lending markets, they might be slower than other vault suppliers, and therefore incur a loss.

8.2 Deposit Can Revert Because Of Market Supply Limit

Note Version 1

CS-RESCRV-002

In `CurveLendOperator`, `_setMintLimit()` calls the `deposit()` function of the market, which can revert if the market's supply limit has been reached.

In the Curve Vault implementation, the `deposit()` function ensures that the total amount of assets held in the vault can not exceed the maximum supply:

```
assert total_assets + assets <= self.maxSupply, "Supply limit"
```

As `_setMintLimit()` is called in `CurveLendOperator`'s `initialize()`, this could lead to `addMarketOperator()` reverting in `CurveLendMinterFactory`, and the deployment of a new `CurveLendOperator` failing.

A revert is more likely to happen if the `CurveLendOperator` is initialized with a high `_initialMintLimit`.

A reverting proposal created by Curve governance cannot be cancelled, and will stay in an "executable" state indefinitely. It might become executable at unforseable times, potentially conflicting with other proposals.

8.3 Deposits Incur Rounding Losses

Note Version 1

When CurveLendOperator deposits crvUSD into a market, the deposited amount credited to the operator can round down by the value of 1 wei of shares. The value of 1 wei of shares can be inflated by an attacker, however, because of the DEAD_SHARES mechanism of Curve lending Vaults, the attacker has to invest 1000 times (the number of DEAD_SHARES) the value that can be stolen from a single deposit. This makes the attack unprofitable unless deposits are repeated more than 1000 times before the attack is noticed.

8.4 Donations to the CurveLendingOperator

Note Version 1

Direct donations can be performed in favor of the CurveLendingOperator. Either Vault shares can be donated to the contract, or crvUSD can be donated which is converted to shares on the next `mintLimit` increase. The donated shares are treated by the operator the same way as yield. Donations can be used to increase the profit transferred by `withdraw_profit()`.