PUBLIC

Code Assessment

of the SparkLend Advanced Smart Contracts

March 12, 2025

Produced for

by

Contents

1 Executive Summary

Dear all,

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

SparkDAO has implemented seven new price oracles to be used within the SparkLend protocol: a fixed price oracle, an oracle with a capped price, a wstETH oracle, a rETH oracle, a weETH oracle, a rsETH oracle and an ezETH oracle. Additionally, a MorphoUpgradableOracle has been introduced for use in MorphoBlue. Furthermore two interest rate strategies have been implemented. One sets the base rate using a rate source, while the other targets a specific rate at optimal utilization. Finally a capped fallback rate source has been added.

The most critical subjects covered in our audit are functional correctness and precision of arithmetic operations. Security regarding all the aforementioned subjects is high.

It is known that the weETH oracle introduced can be prone to manipulation. However, it is deemed that there is no practical risk with the intended configuration. For more information, please refer to issue weETH oracle manipulation

Note that the oracles for wstETH, rETH and weETH may fail to provide accurate results in case of an LST/LRT depeg, see the note ETH oracle is used for LST pricing.

The general subjects covered are specification, gas efficiency, and trustworthiness. Security regarding all the aforementioned subjects is high.

In summary, we find that the codebase provides a good 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	2
• Code Corrected	1
• Risk Accepted	1
Medium-Severity Findings	1
• Risk Accepted	1
Low-Severity Findings	1
• Code Corrected	1

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 SparkLend Advanced 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	11 Dec 2023	2980932a2bed29387eaa9f43cda06c458ffd5e50	Initial Version
2	19 Dec 2023	2835e3f09efe4f2dcf880e23427b5318fbc5a2d0	Extended Scope
3	03 Jan 2024	277ea9d9ad7faf330b88198c9c6de979a2fad561	Fixes
4	04 Mar 2024	40df4afd20028e4ec4cd5be8382934727bb85e06	stETH and rETH Oracles
5	26 May 2024	4d44ee23adf28e59f5405c2a3837b19ab1a84d31	weETH and Morpho Oracles
6	10 Jun 2024	4b3e93869b061123ba86e602466f879817315011	CappedFallbackRateSource
7	11 Jun 2024	57b437239050b3b4862db91cde4568b7b38b7f2e	Fixes
8	29 Nov 2024	58d74302020f07811e79d077b6126c8557d9d310	SUSDS Rate Source
9	12 Feb 2025	338c37115aff4c75a481a6850b0a0594e465573a	rsETH and ezETH Oracles

For the solidity smart contracts, the compiler version `0.8.20` and EVM version `shanghai` was chosen. Since version 9, the compiler version was updated to `0.8.25`, and the EVM version was set to `cancun`.

The following contracts are in the scope of this review:

```
CappedOracle.sol
FixedPriceOracle.sol
```

Starting with Version 2, the scope has been extended to include the following contracts:

```
interfaces/IRateSource.sol
PotRateSource.sol
RateTargetBaseInterestRateStrategy.sol
RateTargetKinkInterestRateStrategy.sol
VariableBorrowInterestRateStrategy.sol
```

Starting with Version 4, the scope has been extended to include the following contracts:

```
RETHExchangeRateOracle.sol
WSTETHExchangeRateOracle.sol
interfaces/IPriceSource.sol
```

Starting with Version 5, the scope has been extended to include the following contracts:

```
WEETHExchangeRateOracle.sol
MorphoUpgradableOracle.sol
interfaces/AggregatorV3Interface.sol
```

Starting with Version 6, the scope has been extended to include the following contract:

```
CappedFallbackRateSource.sol
```

Starting with Version 8, the scope has been extended to include the following contract:

```
SSRRateSource.sol
```

Starting with Version 9, the scope has been extended to include the following contract:

```
EZETHExchangeRateOracle.sol
RSETHExchangeRateOracle.sol
```

2.1.1 Excluded from scope

Any file not explicitly listed above as well as third-party libraries are out of the scope of this review.

2.2 System Overview

In the findings section, we have added a version icon to each of the findings to increase the readability of the report.

SparkLend Advanced offers a suite of features enhancing security and streamlining the automation of governance processes.

2.2.1 SparkLend Oracles

The contracts described in this section are oracles intended to be used within the SparkLend protocol. The oracles have the following interface:

```
interface IOracle {
    function latestAnswer() external view returns (int256);
    function decimals() external pure returns(uint8);
}
```

The price returned is an integer.

Capped Oracle

This oracle returns the minimum price with 8 decimals between a `maxPrice` set during deployment and the price returned by the `source`, which is assumed to have 8 decimals. The price can be queried with `latestAnswer()`. Note that the consumer of this price feed should be able to handle negative prices.

Fixed Price Oracle

This oracle returns a constant price set during deployment with 8 decimals. The price can be queried with `latestAnswer()`.

wstETH Oracle

In Version 4, an oracle for wstETH is introduced. Namely, `latestAnswer()` provides a price that is the product of the ETH price (Chainlink ETH/USD oracle) and Lido's `getPooledEthByShares()` (amount of ETH per share) function. Note that the result is scaled to have 8 decimals. In case, either price feed used returns a number smaller or equal to zero, zero is returned.

rETH Oracle

In Version 4, an oracle for rETH is introduced. Namely, `latestAnswer()` provides a price that is the product of the ETH price (Chainlink ETH/USD oracle) and Rocket Pool's `getExchangeRate()` (amount of ETH per rETH) function. Note that the result is scaled to have 8 decimals. In case, either price feed used returns a number smaller or equal to zero, zero is returned.

weETH Oracle

In Version 5, an oracle for weETH is introduced. Namely, `latestAnswer()` provides a price that is the product of the ETH price (Chainlink ETH/USD oracle) and EtherFi's `getRate()` (amount of ETH per weETH) function. Note that the result is scaled to have 8 decimals. In case, either price feed used returns a number smaller or equal to zero, zero is returned.

ezETH Oracle

In Version 9, an oracle for ezETH is introduced. Namely, `latestAnswer()` provides a price that is the product of the ETH price (Chainlink ETH/USD oracle) and the conversion rate of ezETH:ETH (`RestakeManager.calculateTVLs()` divided by ezETH's total supply). Note that the result is scaled to have 8 decimals. In case, either the computed conversion rate of ezETH:ETH or the Chainlink ETH/USD oracle price is a number smaller or equal to zero, zero is returned.

rsETH Oracle

In Version 9, an oracle for rsETH is introduced. Namely, `latestAnswer()` provides a price that is the product of the ETH price (Chainlink ETH/USD oracle) and KelpDAO LRT Oracle's `rsETHPrice()` (amount of ETH per rsETH) function. Note that the result is scaled to have 8 decimals. In case, either price feed used returns a number smaller or equal to zero, zero is returned.

2.2.2 Morpho Oracles

The contracts described in this section are oracles intended to be used within the Morpho protocol. These oracles are intended to be used by MorphoChainlinkOracleV2 (deployed through corresponding factory) so that it can be used within Morpho Blue. To be compatible with the oracle usage defined in Morpho's ChainlinkDataFeedLib, the oracles implement the functions

1. `latestRoundData`
2. `decimals`

Morpho Upgradeable Oracle

In Version 5, an upgradeable oracle has been introduced. Note that the interface functions wrap the underlying Chainlink price feed's corresponding functions. However, `latestRoundData` only returns the reported `answer` along with zeros for the other return values. Further, note that the underlying price feed can be updated with `setSource()` by the owner of the oracle.

2.2.3 Rate Sources

PotRateSource

This adapter converts the DSR (DAI Savings Rate, per-second compound interest factor in RAY) querried from the POT into APR (Annual Percentage Rate in RAY).

CappedFallbackRateSource

In Version 6, a capped fallback rate source has been introduced. This source, intended for ETH markets, wraps another rate source. It implements upper and lower bounds to cap the rate and includes a fallback mechanism to return a default rate if the primary source query fails. Note that the expected failure to occur is that the source unwhitelists the rate source (leading to reverts).

2.2.4 Custom Interest Rate Strategies

VariableBorrowInterestRateStrategy

Modified version of Aave's `DefaultReserveInterestRateStrategy` with the stable borrow logic removed.

The assumption made is that the stable borrow functionality remains inactive and therefore, its associated functions are not expected to be used. The majority of features connected to the stable borrow functionality have been effectively deactivated by always returning a value of zero. An exception exists for `getBaseStableBorrowRate()`. In order to align closely with the default version, this function returns the value corresponding to variable slope 1.

`calculateInterestRates()` implements the calculation of the interest rates depending on the passed reserve state. The function returns the `liquidityRate` (the supplyRate), the `stableBorrowRate` (present due to the requirements of the interface, always zero since this functionality has been removed) and the `variableBorrowRate`.

The `currentVariableBorrowRate` is determined based on the borrow usage ratio and the 2-slope interest rate model which adjusts the rate depending on whether the borrow usage ratio exceeds the optimal usage ratio.

The `currentLiquidityRate` in this function depends on the current variable borrow rate, the supply usage ratio, and the reserve factor of the protocol.

RateTargetBaseInterestRateStrategy

Derives from the `VariableBorrowInterestRateStrategy` and modifies `_getBaseVariableBorrowRate()` to align with an external rate source (`Rate_source.getAPR()`), with the addition of a constant spread. This strategy is expected to be used for the DAI market, with the external rate source being the annualized DSR.

In Version 4, `getBaseVariableBorrowRateSpread()` has been added as a getter for the constant spread.

RateTargetKinkInterestRateStrategy

Derives from the `VariableBorrowInterestRateStrategy` and modifies `_getVariableRateSlope1()` to set the variable slope 1 rate to match an external rate source (`Rate_source.getAPR()`) and a constant spread. This strategy is expected to be used for ETH, it is expected to track the staked ETH yield minus some spread.

In Version 4, `getVariableRateSlope1Spread()` has been added as a getter for the constant spread.

In Version 8, the SSRRateSource has been added. Similar to the PotRateSource, it converts the SSR (SUSDS Savings Rate, per-second compound interest factor in RAY) querried from the SUSDS into APR (Annual Percentage Rate in RAY).

2.2.5 Trust Model

There is no privileged role in these contracts. The origin of the information (pricing, rates), namely the base oracle or the `Pot` (and SUSDS added in version 8) are completely trusted.

In version 5, the Morpho upgradeable oracle was introduced which has an owner. The owner of the oracle is expected to be the governance and is fully trusted as it could set arbitrary contracts as price sources on the managed Morpho markets.

In version 9, oracles interacting with the Renzo Restake Manager and KelpDAO LRT Oracle contracts were introduced. These external implementations are deployed behind proxies controlled by respective proxy admins. The admins of both contracts are fully trusted as they could update the contracts to arbitrary implementations. It is assumed the underlying token accounted by Renzo ezETH and KelpDAO rsETH are non-manipulatable. In addition:

- It is assumed Renzo's RestakeManagerAdmin is trusted. Otherwise, it can inflate the ezETH:ETH rate by adding malicious operator delegator (`addOperatorDelegator()`). Generally the RestakingManager's TVL calculation must be trusted. These are the inherent risks of Renzo itself, since the ezETH token minting on Renzo also computes the same conversion rate (TVL/totalSupply) as the SparkLend Pricefeed does.
- It is assumed keepers will keep the KelpDAO's rsETH price fresh by periodical updates (`updateRSETHPrice()`). In case the price update fails due to exceeding the percentage limit in an update, the LRT admin, who has the privilege to update the percentage limit, is trusted to react in time to ensure the price is up-to-date.
- It is assumed the burner of both protocol's restaked tokens are trusted. Otherwise, they may decrease the total supply by burning hence inflating the conversion rate.

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
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

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

Critical-Severity Findings	0
High-Severity Findings	1

- weETH Oracle Manipulation Risk Accepted

Medium-Severity Findings	1

- rsETH Price Updates Manipulation Risk Accepted

Low-Severity Findings	0

5.1 weETH Oracle Manipulation

Security High Version 1 Risk Accepted

CS-SPRKADV-001

The weETH oracle can be manipulated upwards by burning eETH. Ultimately, such manipulations may lead to draining the protocol.

Consider that EtherFi will compute the weETH/eETH rate as

```
function amountForShare(uint256 _share) public view returns (uint256) {
    uint256 totalShares = eETH.totalShares();
    if (totalShares == 0) {
        return 0;
    }
    return (_share * getTotalPooledEther()) / totalShares;
}
```

where `_share` is `10**18`. Note that `totalShares()` can be reduced by arbitrary users by burning eETH:

```
function burnShares(address _user, uint256 _share) external {
    require(msg.sender == address(liquidityPool) || msg.sender == _user, "Incorrect Caller");
    ...
    totalShares -= _share;
    ...
```

Consequently, the rate can be increased by decreasing the total shares.

Consider now a scenario where a user borrows 1M USD worth in weETH with 1.5M USD worth in ETH collateral. If the borrower managed to increase the rate by a factor of 2, his position would immediately become unhealthy. Ultimately, that would generate bad debt for Spark.

Note that such an attack requires a donation to EtherFi and could be made more profitable by leveraging high-value positions, especially those that are highly leveraged.

Risk Accepted:

Spark acknowledged the issue but deems it no practical risk given the LTV parameters for weETH Collateral on Sparklend Mainnet and borrowing being disabled. The oracle may be replaced in the future if deemed necessary for higher LTV configurations.

The intended configuration on Github.

5.2 rsETH Price Updates Manipulation

Security Medium Version 9 Risk Accepted

CS-SPRKADV-002

In KelpDAO's LRTOracle, the conversion rate of rsETH:ETH `rsETHPrice` can be updated permissionlessly with `updateRSETHPrice()`. In addition, the delta of the old and new price is restricted to be within a limit, otherwise, the update will revert. By the time of this review (February 2025), the limit is set to 1%.

Theoretically, one can sandwich the rsETH price update (within the limit) with other operations on Spark Lend to arbitrage.

In addition, if the price is not updated in time and increases / decreases more than 1% abruptly, it requires admin privileges to adjust the limit and update the price. Consequently, before the admin operations, a stale price will be used on Spark Lend.

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	1

- Decimals Mismatch in CappedOracle Code Corrected

Medium-Severity Findings	0
Low-Severity Findings	1

- Try-Catch Error Validation Code Corrected

Informational Findings	1

- Gas Optimizations Code Corrected

6.1 Decimals Mismatch in `CappedOracle`

Correctness High Version 5 Code Corrected

CS-SPRKADV-007

The function `CappedOracle.decimals` returns `8` regardless of the decimals of the `source`, that is expected to be a Chainlink price oracle. This creates a decimals mismatch when the source is an `ETH` pair. For example, the Chainlink price oracle for `stETH/ETH` has 18 decimals (https://etherscan.io/address/0x86392dc19c0b719886221c78ab11eb8cf5c52812#readContract#F3). If the price of an `ETH` pair is computed with the decimals of the `CappedOracle`, the price will be either heavily over-evaluated if `maxPrice` has `18` decimals, or always capped to `maxPrice` if it has `8` decimals.

Code corrected:

The oracle is designed to work with source oracles that have 8 decimals. An additional check has been added to the constructor to enforce this 8-decimal precision in the source oracle, preventing possible misconfiguration and resulting decimal mismatch.

6.2 Try-Catch Error Validation

Design Low Version 6 Code Corrected

CS-SPRKADV-006

The `CappedFallbackRateSource` falls back to a default rate when the static call `getAPR` in unsuccessful. The fallback works by wrapping `getAPR` in a `try-catch` block as follows:

```
try source.getAPR() returns (uint256 rate) {
    ...
```

```
    return rate;
} catch {
    return defaultRate;
}
```

Note that `getAPR` may revert due to errors raised (e.g. pause of `source`) or due to out-of-gas scenarios. Note that the code above does not distinguish the scenarios. Thus, out-of-gas is treated as a regular failure. However, note that such scenarios are triggerable by arbitrary addresses.

In general, such a scenario is unlikely. However, the execution flow might be provoked to enter such a path. That will leave the rest of the computation with 1/64th of the gas available before the static call. The remaining gas might be sufficient in some scenarios to complete the call. Namely, if the source can run out of gas with 63x gas, then x gas will be available for the remainder of the execution. If x is high enough, the capped rate source may be manipulated. Note that such attacks are non-trivial and not necessarily feasible. An attacker could prewarm storage slots and addresses to reduce the cost of the remainder of the execution. Further, gas costs might change and lead to potentially dangerous scenarios.

Last, while we expect `getAPR` to only revert in reasonable scenarios, it might be possible that there are manipulations on `source` that will lead to reverts and thus to `defaultRate`.

Ultimately, regular reverts are not distinguished from out-of-gas scenarios. That may lead to the possibility of returning the default rate incorrectly.

Code corrected:

A check was introduced to validate that the error message has a length greater than zero. While that may also include other scenarios, these other scenarios are not expected to occur in the source.

6.3 Gas Optimizations

Informational | Version 1 | Code Corrected

CS-SPRKADV-005

1. The `vars.totalDebt = params.totalStableDebt + params.totalVariableDebt;` computation can ignore the total stable debt, as it should always be 0.

Code corrected:

`totalDebt` has been removed and instead `params.totalVariableDebt` is now used directly.

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 ezETH Price Manipulation

Informational **Version 9**

CS-SPRKADV-003

When computing the ezETH:ETH conversion rate, the Renzo StakeManager compute the TVLs by iterating over existing Operator Delegator and collateral tokens.

Theoretically, if the StakeManager's admin is malicious, it can add an Operator Delegator with compliant interfaces that reports fake amount of assets balance. This will inflate the TVLs immediately and hence the ezETH:ETH conversion rate.

Generally the RestakingManager's TVL calculation must be trusted. These are the inherent risks of Renzo itself, since the ezETH token minting on Renzo also computes the same conversion rate (TVL/totalSupply) as the SparkLend Pricefeed does.

7.2 Inconsistent `setSource()` Event Emission

Informational **Version 5**

CS-SPRKADV-004

`setSource()` emits the `SourceChanged` event before performing the state change which is inconsistent with the constructor's behavior where the state change occurs first. Additionally, `setSource()` emits the event even when setting the source to its current address, which can be confusing due to the event's name `SourceChanged`.

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 ETH Oracle Is Used for LST Pricing

Note Version 1

Users should be aware that the oracles for LSTs (e.g. rETH and wstETH) and LRTs (e.g. weETH) will not handle depeg scenarios by design.

More specifically, the oracles use the ETH/USD price feed for estimating the value of the tokens. However, the LSTs could depeg so that the market may offer the LSTs/LRTs at a discount (e.g. due to long waiting times for withdrawals).

Consequently, pricing the LSTs and LRTs with a ETH/USD oracle may introduce a systematic risk. For example, some users may not be liquidated which could lead to protocol losses.

While severe depegs are rather unlikely, they may happen under extreme conditions. Ultimately, the ETH/USD oracle could lead to inaccuracies.

8.2 Morpho Upgradeable Oracle

Note Version 5

`MorphoUpgradeableOracle` follows the assumptions made on Chainlink oracles in Morpho (e.g. no staleness). Essentially, no checks on the data provided by Chainlink are made which may introduce some risk to the managed markets. Governance is expected to be able to react quickly on Morpho markets.

Further, the only meaningful data returned by `latestRoundData` is the `answer`. All other values are not meaningful (zero values).