Elara Finance

Executive Summary

This audit report was prepared by Quantstamp, the leader in blockchain security.

Type	DeFi Lending Protocol
Timeline	2024-07-30 through 2024-09-06
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	None
Source Code	• ElaraFinance/elara-contracts #35c0b18
Auditors	• Julio Aguilar Auditing Engineer • Valerian Callens Senior Auditing Engineer • Gereon Mendler Auditing Engineer

Documentation quality	Undetermined
Test quality	Undetermined
Total Findings	16 Fixed: 7 Acknowledged: 8 Mitigated: 1
High severity findings	1 Fixed: 1
Medium severity findings	4 Fixed: 1 Acknowledged: 3
Low severity findings	7 Fixed: 5 Acknowledged: 2
Undetermined severity findings	0
Informational findings	4 Acknowledged: 3 Mitigated: 1

Summary of Findings

Elara is a DeFI lending protocol intended to operate on the Zircuit network. The codebase is a fork of Compound V2, with some modifications. The primary modifications introduced include the addition of a new set of Oracle contracts that support Chainlink, Pyth, and API3. Furthermore, the project team has implemented support for both native and wrapped Ether within a single contract `CEtherV2.sol`, where WETH acts as the underlying asset. Finally, it is now possible to define specific supply caps for the supported assets.

However, the project has no documentation per se and there is also no test suite, which raises concerns about the reliability of the new changes and features. Also, the auditing team found several issues from Info to High severity. Addressing these issues will help ensure the project is stable and secure before launch.

Fix-Review Update: The client has addressed all issues by either fixing, mitigating, or acknowledging them. Additionally, they informed us that the test suite is currently under development and will be completed in the near future. We would also like to commend the client for being highly communicative throughout the process.

ID	DESCRIPTION	SEVERITY	STATUS
ELA-1	Usage of Manually Updated Prices Could Result in Protocol Insolvency in Case of Low Update Frequency and Significant Price Deviation	High	Fixed
ELA-2	Out-of-Gas Errors Can Occur if Too Many Markets Are Supported or Entered	Medium	Acknowledged
ELA-3	Manually Updated Prices Could Be Outdated	Medium	Fixed
ELA-4	The Inheriting Oracles Can Accept Outdated Prices	Medium	Acknowledged
ELA-5	Test Suite Missing	Medium	Acknowledged

ID	DESCRIPTION	SEVERITY	STATUS
ELA-6	Lack of Paginating Functions in `CompoundLens` to Iterate over the Current Borrowers	Low	Fixed
ELA-7	If a User Is the Last Borrower of the List `allBorrowers`, Completely Repaying All Its Debts Will Revert	Low	Fixed
ELA-8	`BaseJumpRateModelV2` Contract Incorrectly Assumes that Blocks Are Produced Every Three Seconds on the Target Network, Instead of Two	Low	Acknowledged
ELA-9	Tokens Marked as Same Can Have Different Direct Prices	Low	Fixed
ELA-10	Missing Input Validations	Low	Acknowledged
ELA-11	The `BaseOracle` Contract Could Return Zero Prices	Low	Fixed
ELA-12	Cannot Remove First Item From the `allBorrowers` List	Low	Fixed
ELA-13	Incorrect Contract Used to Update Calculations	Informational	Acknowledged
ELA-14	Application Monitoring Can Be Improved by Emitting More Events	Informational	Mitigated
ELA-15	Borrower Tracking Does Not Consider Liquidations	Informational	Acknowledged
ELA-16	Aspects to Consider Regarding the `Multicall` Contract to be Aware Of	Informational	Acknowledged

Assessment Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

Disclaimer

Only features that are contained within the repositories at the commit hashes specified on the front page of the report are within the scope of the audit and fix review. All features added in future revisions of the code are excluded from consideration in this report.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

1. Code review that includes the following
    1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
    2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
    3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.

2. Testing and automated analysis that includes the following:
    1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
    2. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarity, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Scope

The scope involves all smart contracts in the `contracts` directory.

Files Included

- Governance/Comp.sol
- Lens/CompoundLens.sol
- Oracle/
    - Api3Oracle.sol
    - BaseOracle.sol
    - ChainlinkOracle.sol
    - HomoraMath.sol
    - PythOracle.sol
- Utils/
    - Multicall.sol
    - SafeMath.sol
    - Timelock.sol
- BaseJumpRateModelV2.sol
- CErc20.sol
- CErc20Delegate.sol
- CErc20Delegator.sol
- CErc20Immutable.sol
- CEther.sol
- CEtherV2.sol
- CEtherV2Delegate.sol
- CEtherV2Delegator.sol
- CEtherV2Immutable.sol
- Comptroller.sol
- CTokenInterfaces.sol
- EIP20Interface.sol
- EIP20NonStandardInterface.sol
- ErrorReporter.sol
- ExponentialNoError.sol
- InterestRateModel.sol
- JumpRateModelV2.sol
- Maximillion.sol
- PriceOracle.sol
- Unitroller.sol

Operational Considerations

- **Admin Role:** The system assumes that the admin role is securely managed, with only trusted individuals having access. Admins have extensive control over the system, including critical parameters, pausing operations, and managing roles.
- **Collateral Factors:** Each market has a collateral factor, determining the maximum borrowing capacity against collateral. These factors are assumed to be conservatively set to minimize the risk of under-collateralization.
- **Upgradeability:** The admin can update implementation addresses for delegator contracts. We assume that rigorous testing and deployment processes are followed.
- **Timelock Contract:** A Timelock contract is used to manage time-delayed execution of critical transactions. It is assumed that the delay period allows enough time for community review and intervention when necessary.
- The team is aware of the Empty Market Vulnerability present in Compound V2-like codebases and knows how to deploy new markets in a way that such an attack is not possible.
- The privileged roles of the system and the associated privileges should be documented to users. Also, key management should follow the latest best practices in terms of security.
- When updating core protocol parameters, the four-eyes principle could be enforced to make sure that no incorrect data is submitted on-chain. For instance, if the admin role in the Oracle contracts sets an incorrect price feed for a supported asset via the function `updatePriceFeed()`, all operations involving fetching the price of a specific asset will revert.

Key Actors And Their Capabilities

- The admin of the system has the following capabilities
  - Set and update various parameters (e.g., interest rate models, reserve factors, price oracles).
  - Pause and unpause certain actions (e.g., minting, borrowing, transferring).
  - Add and remove markets.
  - Manage admin roles and pending admin roles.
  - Manage `COMP` distribution speeds.
  - Manage reserves (add and reduce).
  - Manage implementation addresses for delegator contracts.
  - Manage price feeds for oracles.
  - Execute queued transactions in the `Timelock` contract.
- The `CapGuardian` role in the `Comptroller` contract can update the supply and borrow caps of supported `cTokens`.

Findings

ELA-1

Usage of Manually Updated Prices Could Result in Protocol Insolvency in Case of Low Update Frequency and Significant Price Deviation

High Fixed

Update

The client has resolved the issue by giving priority to Oracle price feeds over manually updated prices. However, the `BaseOracle` contract still includes the manual pricing functionality. It's important to note that the team clarified this contract is meant for testing purposes only. The production contracts, such as `Api3Oracle`, will be used in live environments and no longer exhibit this issue.

Update

Marked as "Fixed" by the client.
Addressed in: `12cbe0a9f0efaa2dfe8a2577e33102a40802f584` and `e122f9e32434c7543b9d84756ca4ff2ad4786dde`.
The client provided the following explanation:

```
We have updated our system to rely entirely on oracle price feeds for stablecoins, eliminating the use of manually updated prices. Besides, to avoid price deviation, we add price upper limit for stablecoins and LRT/LSTs, ensuring their price feed will not exceed their intrinsic value. For example, UpperLimit of $USDC = $1 ; UpperLimit of $weETH  = 1.1 $ETH
```

File(s) affected: `BaseOracle.sol`, `Api3Oracle.sol`, `ChainlinkOracle.sol`, `PythOracle.sol`

Description: When manually updating prices, the admin must send a transaction to the system, which opens the door to potential front-running. If the update frequency is too low and a high price deviation happens, a user could manage to borrow assets before the price is updated from a lower to a higher value, and the system could overestimate the collateral's worth, leading to the accumulation of bad debt. Repeated and opportunistic manipulation of these price updates could destabilize the lending platform, potentially resulting in financial losses for the protocol and users.

Furthermore, it is possible to perform operations involving significant amounts of assets using flash loans to benefit from this price deviation, with the consequences of overpriced collateral and underpriced borrowed amounts.

In one of the worst-case scenarios, if a stablecoin loses its peg and the system has not yet reflected the updated price, users could borrow against their devalued collateral and default on their loans, further increasing the risk of bad debt accumulation.

Exploit Scenario:
Bad Debt Scenario:

- 1 WETH is worth 1000 USD.
- The latest update of the direct price states that 1 USDC is worth 1 USD. However, due to market conditions, the price dropped by 10% and it actually is worth 0.9 USD.
- The Collateral Factor of USDC is 95%.
- Bob has 10k USDC as collateral, so he can borrow a maximum of assets worth 9.5k USD.
- Bob borrows WETH worth 9.4k USD before the USDC price gets updated from 1 USD to 0.9 USD.
- After the update, Bob's collateral is now worth 9k USD for the system and what he borrowed is now worth 9.4k USD, which is a situation of under-collateralization or bad debt creation.

Unintentional Front-Running Scenario:

- 1 WETH is worth 2000 USD.
- The system believes that 1 USDC is worth 1 USD. However, due to market conditions, the price dropped 10% and it actually is worth 0.9 USD.
- The collateral Factor of USDC is 50%.
- User A has 20k USDC as collateral.
- The user manages to borrow 10k USDC before the USDC price gets updated from 1 USD to 0.9 USD.

- The user balance would be 20k USDC as collateral and 5 WETH borrowed. The system believes this is equivalent to 20k USD and 10k USD respectively. However, after the update, the value is actually worth 18k USD and 10k USD which would put the user's position underwater and be ready to be liquidated.
- This might not be the desired outcome of the user, if they had no ill intentions, and just wanted to borrow the maximum amount possible.
- The same issue also results from having WETH as collateral and borrowing USDC, with a depeg in the opposite direction.

Recommendation: We recommend using Oracle price feeds only and not relying on manually updated prices at all.

ELA-2
Out-of-Gas Errors Can Occur if Too Many Markets Are Supported or Entered

Medium | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
We understand and appreciate the concern raised in the audit report regarding potential Out-of-Gas
errors. However, we have decided to retain the original Compound code for the following reasons:
1. Historical Performance: Compound has not experienced such Out-of-Gas errors in practice.
2. Stress Testing: We have conducted extensive stress tests on other blockchains, including
   interactions with over 200,000 wallet addresses, without encountering this issue.
3. Practical Limitations: In our actual operations, we will strictly control the number of listed
   assets, limiting the maximum to 4. This significantly reduces the risk of reaching gas limits.
```

File(s) affected: `Comptroller.sol`

Description: The list `allMarkets` in `ComptrollerV3Storage` records all CTokens supported by the system. This list is iterated when a user wants to claim `COMP` from all its entered markets via the function `Comptroller.claimComp()`, and when a new market is added in `Comptroller._addMarketInternal()`. If too many markets are supported, it will not be possible to add a new one, and the function `claimComp()` will also start reverting because the amount of gas used is too high.

Also, the `maxAssets` value is defined in `ComptrollerV1Storage` to cap the maximum number of markets a specific address can enter as a borrower. However, this maximum cap is not used in the code. As a result, if a lot of markets are supported and the user enters all of them, it could avoid the execution of some operations such as liquidations if they consume too much gas when browsing the list of assets he borrowed (`ComptrollerV1Storage.accountAssets`).

Recommendation: While this could become a problem when too many markets are supported, we recommend identifying worst-case scenarios by performing gas simulations to identify what is the maximum amount of markets that should be supported by the system (`maxSup`). Same recommendation for the number of markets a user can enter (`maxEnt`). If `maxEnt < maxSup`, consider enforcing checks based on the value of `maxAssets`.

ELA-3 Manually Updated Prices Could Be Outdated

Medium | Fixed

Update

The client has resolved the issue by giving priority to Oracle price feeds over manually updated prices. However, the `BaseOracle` contract still includes the manual pricing functionality. It's important to note that the team clarified this contract is meant for testing purposes only. The production contracts, such as `Api3Oracle`, will be used in live environments and no longer exhibit this issue.

Update

Marked as "Fixed" by the client.
Addressed in: `12cbe0a9f0efaa2dfe8a2577e33102a40802f584`.
The client provided the following explanation:

```
We have adopted the recommendation from the audit report and updated our system to use oracle price
feeds for all assets, including stablecoins. This change addresses the issues raised in the report:
1. Eliminates the risk of outdated prices due to manual updates.
2. Avoids system risks that could arise from administrators not updating prices in a timely manner.
3. Ensures the timeliness and accuracy of all price data.
```

File(s) affected: `BaseOracle.sol`, `Api3Oracle.sol`, `ChainlinkOracle.sol`, `PythOracle.sol`

Description: The Oracle contracts are meant to use manually updated prices for stablecoins as a primary data feed. They can be updated at any time by the admin, however, when fetching the prices, there is no way to check if they were last updated within a reasonable time window. This could lead to potentially using old prices.

Recommendation: Ideally, even stablecoins should use Oracle feeds instead of manually updated prices. However, if that solution is not desired, we recommend adding a second mapping to store the last updated timestamps and only accept prices that were updated within a reasonable time window, or return 0 if that is not the case.

ELA-4 The Inheriting Oracles Can Accept Outdated Prices

Medium | Acknowledged

Update

Marked as "Fixed" by the client.
Addressed in: `12cbe0a9f0efaa2dfe8a2577e33102a40802f584`.
The client provided the following explanation:

```
We set maxDelayTime when deploying the contract, rather than initializing it in the constructor. We have currently set maxDelayTime to one month on our deployed contracts on Zircuit testnet. When deploying on the Zircuit mainnet, we will ensure that maxDelayTime is greater than the heartbeat of the oracle price feed. (e.g. 1 month)
```

File(s) affected: `API3Oracle.sol`, `ChainlinkOracle.sol`, `PythOracle.sol`

Description: The `API3Oracle`, `ChainlinkOracle`, and `PythOracle` implementations are at risk of accepting outdated prices because the `maxDelayTime` variable is not initialized during construction. As a result, this variable could remain unset for an indefinite period, potentially causing these oracles to deliver outdated price information.

Recommendation: We recommend initializing `maxDelayTime` to a non-zero value in the constructor. Also, for more information about integrating these Oracles, see the following:

- Pyth oracles, keep an eye on the best practices described here: https://docs.pyth.network/price-feeds/best-practices
- API3 oracles, keep an eye on the best practices described here: https://docs.api3.org/explore/dapis/security-considerations.html
- Chainlink oracles, keep an eye on the best practices described here: https://docs.chain.link/data-feeds/developer-responsibilities

ELA-5 Test Suite Missing

Medium | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
Still working on it.
  But we have some challenges in Adapting Compound's Test Suite
  Environment Mismatch: We use Hardhat; Compound uses proprietary 'Saddle'. Direct migration is unfeasible.
  Industry Norm: Other projects (e.g., Orbit Lending) using Hardhat also haven't integrated Compound's tests.
  Proven Stability: Elara's codebase has demonstrated reliability through real-world deployments.
```

Description: Even though Elara is initially a Compound Fork that has been battle-tested throughout the years, there are some changes made that require testing to ensure the correct behavior of the system. Currently, there is no test suite, and small bugs or unexpected behavior could still be present in the codebase.

Recommendation: We recommend forking and adapting the test suite from Compound and improving on that by including tests that cover the new features introduced by Elara.

ELA-6 Lack of Paginating Functions in `CompoundLens` to Iterate over the Current Borrowers

Low | Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `f6ebde3c56a4e9ad3120e83c9c1eb5a9710b411e`.
The client provided the following explanation:

```
We have addressed this issue by removing the allBorrowers-related code. This elimination resolves the potential gas consumption problems associated with iterating over a large list of borrowers.
  Liquidation Approach:
  In the Elara system, we do not rely on CompoundLens for fetching borrowing data or executing
```

```
liquidation operations. Instead:
  Our official liquidation bot has implemented its own efficient data fetching functionality.
  Third-party bots are also equipped with custom data retrieval mechanisms.
```

File(s) affected: `CompoundLens.sol`

Description: The functions `getAllBorrowersData()`, `getComptrollerData()`, and `getBorrowersData()` iterate over all the current borrowers of the system. As this list can become very long, the iteration may result in an out-of-gas error. In this case, these functions would temporarily become unusable until the list is reduced.

Recommendation: Consider adding an alternative to these functions using pagination to let users easily get information about the system when the list of borrowers becomes too long.

ELA-7

If a User Is the Last Borrower of the List `allBorrowers`, Completely Repaying All Its Debts Will Revert

Low Fixed

Update

Marked as "Fixed" by the client.

Addressed in: `f6ebde3c56a4e9ad3120e83c9c1eb5a9710b411e`.

The client provided the following explanation:

```
Same as ELA-6, allborrowers removed
```

File(s) affected: `Comptroller.sol`

Description: If the user `Alice` borrows assets from the system, her address is added to the list `allBorrowers` in the function `addToMarketInternal()`. When she repays what she borrowed such that she is no longer considered a borrower, the function `exitMarket()` is executed and the address of `Alice` is removed from the list `allBorrowers`. To save gas, this removal uses two additional data structures to swap the address of `Alice` with the end of the list, and then remove the last item of the list. However, if `Alice` is already the last item of the list, an out-of-bound error will happen, as described in the scenario below:

- Initial state:

```
allBorrowers = [Bob, Alice]
borrowers[Bob] = true
borrowers[Alice] = true
borrowerIndexes[Bob] = 0
borrowerIndexes[Alice] = 1
```

- We remove Alice (`msg.sender == Alice`)

```
//We enter in the if condition since Alice index is 1
if (borrowerIndexes[msg.sender] != 0 && storedList.length == 0) {
//Alice is replaced by Alice in allBorrowers
allBorrowers[borrowerIndexes[msg.sender]] = allBorrowers[allBorrowers.length - 1];
// We obtain: allBorrowers = [Bob, Alice]
//Alice is removed from allBorrowers
allBorrowers.pop();
// We obtain: allBorrowers = [Bob]
borrowerIndex[Alice] = 1
borrowerIndex[Bob] = 0
// The following line reverts because we try to access the item of the list allBorrowers at id = 1, which
is out-of-bound
borrowerIndexes[allBorrowers[borrowerIndexes[msg.sender]]] = borrowerIndexes[msg.sender];
```

The impact is limited since the issue can be bypassed by Alice by repaying all her debts except `1 wei`, or by waiting for a new borrower to be added to the end of the list. Still, this situation should be fixed to prevent an unnecessary increase of the list `allBorrowers`.

Recommendation: Consider updating the logic of this mechanism to avoid this out-of-bound error. Also, note that the value of `borrowerIndexes[msg.sender]` can be cached in a local variable to save gas.

ELA-8

`BaseJumpRateModelV2` Contract Incorrectly Assumes that Blocks Are Produced Every Three Seconds on the Target Network, Instead of Two

Low | Acknowledged

Update

Marked as "Fixed" by the client.
Addressed in: `12cbe0a9f0efaa2dfe8a2577e33102a40802f584`.
The client provided the following explanation:

> When deploying on the Zircuit mainnet, we will use an accurate blocksPerYear value calculated based on the network's actual block time

File(s) affected: `BaseJumpRateModelV2.sol`

Description: The contract `BaseJumpRateModelV2` defines the storage variable:

```
/**
 * @notice The approximate number of blocks per year that is assumed by the interest rate model
 */
uint public blocksPerYear = 10512000; // 3 second per block
```

However, since on the target network blocks are produced every two seconds, the expected number of blocks in one year is closer to `15768000` than to `10512000`. The impact is limited though since that variable can be updated via the function `updateBlocksPerYear()`.

Recommendation: Consider updating the hardcoded value and the comment.

ELA-9 Tokens Marked as Same Can Have Different Direct Prices

Low | Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `12cbe0a9f0efaa2dfe8a2577e33102a40802f584`.
The client provided the following explanation:

> Same as ELA-1. Elara will rely entirely on Oracle-provided price data for all assets.

File(s) affected: `BaseOracle.sol`

Description: The `setSameToken()` function allows tokens to be marked as identical, such that the same Oracle data feed is queried for them. However, the direct price mechanism still allows two separate entries.
Token A can use the `sameTokens` mapping to refer to the price feed of another token B instead. This currently only works for Oracle prices, not the manually set direct prices. If such a token A has a direct price set, this value will be prioritized over the mapping entry B, even when the other token B also has a directly assigned, potentially conflicting price. In other words, for oracle-sourced prices the value is taken from B, but for direct prices from A.

Recommendation: Identify the preferred behavior and adjust the code base accordingly. However, we recommend removing both mappings to avoid unexpected behavior.

ELA-10 Missing Input Validations

Low | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

> Acknowledged

File(s) affected: `Api3Oracle.sol`, `ChainlinkOracle.sol`, `PythOracle.sol`, `BaseJumpRateModelV2.sol`, `CToken.sol`, `Comptroller.sol`

Description: It is important to validate inputs, even if they only come from trusted addresses, to avoid human error:

1. The `tokens` and `data` arrays should have the same length in `Api3Oracle.updatePriceFeed()`, `ChainlinkOracle.updatePriceFeed()` and `PythOracle.updatePriceFeed()`;
2. `blocksPerYear` should not be updated with an abnormal value in `BaseJumpRateModelV2.updateBlocksPerYear()`. Consider adding a relevant lower and upper limit.
3. `CToken.initialize()` should make sure the name and symbol are not empty strings and that the decimals are less than 18.
4. `protocolSeizeShareMantissa` should not be updated with an abnormal value in `CToken._setProtocolSeizeShare()`. Consider adding a relevant upper limit.
5. The `ChainlinkOracle` and `PythOracle` should ONLY work with USD price feeds, however, the code does not enforce that. Both oracles assume the decimals of their corresponding price feeds equal `USD_DECIMAL` without validating `decimals()` and `expo` respectively. If a mistake is made, a different scaling would be used, returning the wrong prices.
6. `Comptroller.setCompAddress()` is missing a zero-address check for `newAddress`.
7. `BaseJumpRateModelV2.updateJumpRateModel()` is missing checks for all parameters.
8. `BaseOracle.setMaxDelayTime()` is missing a check for reasonable time values.

Recommendation: Consider adding the relevant checks.

ELA-11 The `BaseOracle` Contract Could Return Zero Prices

Low Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `12cbe0a9f0efaa2dfe8a2577e33102a40802f584`.
The client provided the following explanation:

> Same as ELA-1. We do not directly use BaseOracle. Instead, we rely entirely on oracle price feeds

File(s) affected: `BaseOracle.sol`

Description: The `getOraclePrice()` function always returns zero due to incomplete implementation, which causes the `getPrice()` function to potentially return zero when calling `getPriceByToken()`, as it relies on `getOraclePrice()` if the `cToken` is not in the `directPrices` mapping. The primary function used throughout the codebase to fetch token prices, `getUnderlyingPrice()`, also depends on these two functions, meaning it could return zero prices as well. Fortunately, the `getOraclePrice()` function is overridden in the inheriting contract, where it returns a non-zero value. However, since `BaseOracle` is not an abstract contract, it could theoretically be used directly, allowing this issue to persist.

Recommendation: We recommend removing the implementation of `getUnderlyingPrice()` and `getOraclePrice()` from the `BaseContract` and leaving them as virtual functions to be overridden by the inheriting contract. Additionally, consider making the `BaseOracle` an abstract contract.

ELA-12 Cannot Remove First Item From the `allBorrowers` List

Low Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `f6ebde3c56a4e9ad3120e83c9c1eb5a9710b411e`.
The client provided the following explanation:

> Same as ELA-6, allborrowers removed

File(s) affected: `Comptroller.sol`

Description: The `exitMarket()` function removes users from the `allBorrowers` list if they have exited all the markets. It does so by checking that the corresponding `borrowerIndex` mapping is not zero and that the `storedList` is empty. However, the very first user stored in the list has index zero. As a result, we do not enter the if condition and the first user is not removed from the list, while no longer having any asset borrowed from the system.

Recommendation: A possible solution is to use the `length` instead of `length - 1` when storing the borrower index, and do the subtraction by one when fetching the index from `borrowerIndex`.

ELA-13 Incorrect Contract Used to Update Calculations

Informational Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

Acknowledged

File(s) affected: `BaseJumpRateModelV2.sol`

Description: The function `updateBlocksPerYear()` in the `BaseJumpRateModelV2` calculates `multiplierPerYear` as `multiplierPerBlock * _blocksPerYear`. However, users might expect it to be `(multiplierPerBlock * _blocksPerYear * kink) / BASE` since the same contract uses `kink` and `BASE` to calculate `multiplierPerBlock` in the function `updateJumpRateModelInternal()`.

Recommendation: Consider updating the calculation or moving the function to the `JumpRateModelV2` contract where the overridden function `updateJumpRateModelInternal()` would match the current calculation.

ELA-14
Application Monitoring Can Be Improved by Emitting More Events

Informational | Mitigated

Update

The client implemented some suggestions in commit `e122f9e32434c7543b9d84756ca4ff2ad4786dde`.

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

Acknowledged

File(s) affected: `BaseJumpRateModelV2.sol, Api3Oracle.sol, ChainlinkOracle.sol, PythOracle.sol, BaseOracle.sol, Comptroller.sol`

Description: To validate the proper deployment and initialization of the contracts, it is a good practice to emit events for important state transitions. It would also be beneficial to monitor the contract and track eventual bugs or hacks. Below is a list of events that could be added:

1. New `owner` in `BaseJumpRateModelV2.transferOwnership()`;
2. New `blocksPerYear` in `BaseJumpRateModelV2.updateBlocksPerYear()`;
3. Fixed New `priceFeed` in `Api3Oracle.updatePriceFeed()`, `ChainlinkOracle.updatePriceFeed()` and `PythOracle.updatePriceFeed()`;
4. Fixed New `directPrice`, `sameToken`, `maxDelayTime` and `admin` in `BaseOracle.sol`.
5. New `compAddress` in `Comptroller.setCompAddress()`.
6. `CToken._setProtocolSeizeShare()` can emit events even when `protocolSeizeShareMantissa` has not changed. Evaluate this behavior and update as desired.

Recommendation: Consider emitting the suggested events.

ELA-15
Borrower Tracking Does Not Consider Liquidations

Informational | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

Acknowledged

File(s) affected: `Comptroller.sol`

Description: This protocol aims to track current borrowers in the Comptroller. However, this tracking is only updated when an address enters or exits a market of its own accord, but not if its position is fully liquidated by another user. In this case, while unlikely due to the requirement to have a closing factor of 100%, the address would remain on the list until another operation removes it from the list.

Recommendation: Update this tracking or document this behavior.

ELA-16

Aspects to Consider Regarding the `Multicall` Contract to be Aware Of

Informational | Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> ```
> Acknowledged
> ```

Description:

- The function `blockhash()` will return 0 for a block number older than 256 from the current one.
- The `block.difficulty` built-in in Solidity has been deprecated since the EVM Paris Hardfork.
- Anyone can execute the function `aggregate()` in the `Multicall` contract because the function has no access control restriction. As a result, if this contract owns assets or has privileged roles on third-party contracts, anyone can transfer these tokens or call these third-party contracts.

Recommendation: Carefully consider the use of this contract and make sure to not give it privileged roles.

Auditor Suggestions

S1 Using `transfer()` for ETH May Fail in The Future

Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> ```
> Acknowledged
> ```

File(s) affected: `CEther.sol, CEtherV2.sol`

Description: The function `address.transfer()` assumes a fixed amount of gas to execute the call. The use of this function protects against reentrancy attacks. The default amount of gas, however, may change in the future. In the worst case, it could lead to failed transfers.

Recommendation: We want you to be aware of this possibility. Whereas we do not recommend taking any action now, if you need more flexibility regarding the forwarded gas, you can use `call()` to perform transfers.

S2 Unlocked Pragma

Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> ```
> Acknowledged
> ```

Related Issue(s): SWC-103

Description: Every Solidity file specifies in the header a version number of the format `pragma solidity (^)0.8.*`. The caret (`^`) before the version number implies an unlocked pragma, meaning that the compiler will use the specified version *and above*, hence the term "unlocked".

Recommendation: For consistency and to prevent unexpected behavior in the future, we recommend to remove the caret to lock the file onto a specific Solidity version.

S3 Two-Step Ownership Transfers

Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:

```
Acknowledged
```

File(s) affected: `BaseJumpRateModelV2.sol`, `BaseOracle.sol`

Description: Transferring privileged roles in one step can be error prone, thus it is recommended to follow a two-step process where the recipient needs to actively accept the role.

Recommendation: Add a two-step transfer mechanism or use OpenZeppelin contracts for this purpose.

S4 Missing `NatSpec` Comments

Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
Acknowledged
```

File(s) affected: `Comptroller.sol`

Description: The new features or changes are not always documented. The following is a non-exhaustive list of such cases:

1. In `Comptroller.distributeSupplierComp()`, the new function parameter `supplierTokens` is missing from the NatSpec comment.
2. In `Comptroller.updateCompSupplyIndex()`, the new function parameter `supplyTokens` is missing from the NatSpec comment.
3. `CToken` is missing NatSpec documentation for the freshly introduced `isEth` parameters.

Recommendation: For better code comprehension, consider improving the NatSpec and in-function-body comments of the new features and changes.

S5 Suggestions for Code Conciseness and Gas Optimization

Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
Acknowledged
```

File(s) affected: `BaseOracle.sol`, `CompoundLens.sol`

Description: 1. The variable `nativeSymbol` in the `BaseOracle` contract can be declared immutable to make the contract a bit more gas-efficient.

2. In the function `BaseOracle.getPriceByToken()`, the `if (decimalDelta > 0)` statement can be removed since `(10 ** decimalDelta)` would return one if `decimalDelta` is zero.
3. The `BaseOracle` contract declares the variable `isPriceOracle`, which is already defined in the abstract contract `PriceOracle` used as an interface for invoking Oracle functions. This results in a shadow variable, and as a best practice, it is advisable to remove it.
4. In `CompoundLens.getBorrowerData()`, in the if statement `if (ComptrollerLensInterface(comptroller).checkMembership(borrower, CToken(marketData.cToken)))`, the function casts the `comptroller` variable to the type `ComptrollerLensInterface`. However, since `comptroller` is already of that type, the cast is unnecessary.
5. The `BaseOracle` contract determines if the token to get a price for is the native token in two ways: comparing the symbol strings and the token address. Sometimes it is a good design to perform the same functionality by different means. However, in this case, it is unclear the advantages one has over the other. We recommend choosing only one method to determine if the query token is the native token.
6. `BaseOracle` should inherit from `PriceOracle` since the latter is the one being used in the rest of the codebase to get prices.
7. Since the function `BaseOracle.getPrice()` is not used internally, consider removing it.
8. The `keeper` role in the `BaseOracle` is not used. Consider removing it.
9. `JumpRateModelV2.updateJumpRateModelInternal` uses `BASE` in its calculation despite it canceling out mathematically. Consider removing `BASE` from the calculation.
10. `CompoundLens.getAllBorrowersData()` contains code duplicated from `getComptrollerData()` and can call that function instead.
11. This contract still uses the default Compound name and COMP symbol variables. Consider changing these to Elara-specific identifiers.

Recommendation: Consider applying these suggestions.

S6 Unused Code

Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
Acknowledged
```

File(s) affected: `Multicall.sol`, `HomoraMath.sol`, `Maximilion.sol`, `CEther.sol`

Description: Unused code should be avoided since it might consume more resources during development, testing, and compilation without contributing to the software's functionality. It makes the codebase harder to understand and may mislead developers, leading to potential bugs or wasted time. In the case of solidity, it might also increase the bytecode size increasing deployment costs. The `Multicall.sol`, `Maximilion.sol`, `CEther.sol`, and `HomoraMath.sol` contracts are not used at all within the codebase.

Recommendation: Consider removing these files.

S7 Unnecessary Usage of `SafeMath` Library

Acknowledged

Update
Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
Acknowledged
```

File(s) affected: `SafeMath.sol`, `Timelock.sol`

Description: The `SafeMath` library is no longer necessary since its safety features are now built-in in the solidity compiler version greater or equal to `0.8` and it would result in redundant checks consuming more gas for basic operations.

Recommendation: Consider removing the file `SafeMath.sol`.

S8 Impact of Using Non-Standard `ERC20` Tokens as Underlying Tokens.

Acknowledged

Update
Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
Acknowledged
```

Description: Some ERC20 tokens can be:

- paused.
- upgraded.
- whitelisted or blacklisted for some addresses.
- with a fee mechanism.
- with a rebasing mechanism.
- with a hook mechanism when performing a transfer, which can lead to reentrancies.

Recommendation: The codebase already considers tokens with a fee mechanism. However, when considering the support of a new asset, assess carefully the impact of each of these features on your system.

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
- **Informational** – The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
- **Undetermined** – The impact of the issue is uncertain.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.

- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **Acknowledged** – The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Files

- `e55...a36 ./contracts/EIP20NonStandardInterface.sol`
- `40a...57c ./contracts/ComptrollerInterface.sol`
- `4de...a36 ./contracts/CEtherV2.sol`
- `860...7e5 ./contracts/Comptroller.sol`
- `a34...bef ./contracts/CErc20.sol`
- `b16...110 ./contracts/PriceOracle.sol`
- `3b5...0c0 ./contracts/CToken.sol`
- `ec4...d47 ./contracts/CErc20Delegator.sol`
- `ce5...32f ./contracts/CErc20Delegate.sol`
- `5cc...7a9 ./contracts/ExponentialNoError.sol`
- `73a...ff1 ./contracts/CEtherV2Immutable.sol`
- `31d...f65 ./contracts/ErrorReporter.sol`
- `b39...fe2 ./contracts/Unitroller.sol`
- `5c1...6f7 ./contracts/CEtherV2Delegate.sol`
- `54e...169 ./contracts/EIP20Interface.sol`
- `edb...019 ./contracts/InterestRateModel.sol`
- `3df...5f1 ./contracts/JumpRateModelV2.sol`
- `47a...627 ./contracts/ComptrollerStorage.sol`
- `082...a49 ./contracts/CEtherV2Delegator.sol`
- `0ea...34e ./contracts/CErc20Immutable.sol`
- `31d...e48 ./contracts/CEther.sol`
- `c24...c71 ./contracts/Maximillion.sol`
- `03a...a19 ./contracts/BaseJumpRateModelV2.sol`
- `767...6de ./contracts/CTokenInterfaces.sol`
- `e64...9d5 ./contracts/Governance/Comp.sol`
- `ad2...7df ./contracts/Lens/CompoundLens.sol`
- `ed4...4db ./contracts/Utils/SafeMath.sol`
- `508...6af ./contracts/Utils/Timelock.sol`
- `f36...df8 ./contracts/Utils/Multicall.sol`
- `94f...ba9 ./contracts/Oracle/Api3Oracle.sol`
- `18d...f54 ./contracts/Oracle/PythOracle.sol`
- `c35...91a ./contracts/Oracle/HomoraMath.sol`
- `0ff...540 ./contracts/Oracle/BaseOracle.sol`
- `da9...7b0 ./contracts/Oracle/ChainlinkOracle.sol`
- `ab0...36e ./contracts/Mock/ERC20Mintable.sol`

Automated Analysis

N/A

Changelog

- 2024-09-06 - Initial report
- 2024-09-23 - Final report

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over $200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT

Elara Finance