PROTOFIRE

SMART CONTRACT AUDIT REPORT

Cyclo

Version: 1.0

Protofire
November, 2024

Table Of Contents

The Rain audit report

The audit report was prepared for the **Rain Language Team** and includes following Github repositories and files:

https://github.com/cyclofinance/cyclo.sol and

https://github.com/gildlab/ethgild audited commit

34074b93e912dce11415cb464cc0c8b4f9bd2493 and

e51e3697188907b066d3a127c33d2d8ce46b9f17:

-src/concrete/receipt/CycloReceipt.sol
-src/concrete/receipt/Receipt.sol
-rain.math.fixedpoint/lib/LibFixedPointDecimalArithmeticOpenZeppelin.sol
-rain.math.fixedpoint/lib/format/LibFixedPointDecimalFormat.sol
-rain.math.fixedpoint/lib/FixedPointDecimalConstants.sol
-src/concrete/vault/ERC20PriceOracleReceiptVault.sol
-src/concrete/oracle/SceptreStakedFlrOracle.sol
-src/concrete/oracle/TwoPriceOracleV2.sol
-src/concrete/oracle/FtsoV2LTSFeedOracle.sol
-rain.flare/lib/lts/LibFtsoV2LTS.sol
-src/abstract/PriceOracleV2.sol
-rain.flare/lib/sflr/LibSceptreStakedFlare.sol
-src/abstract/ReceiptVault.sol
-rain.factory/blob/main/src/concrete/CloneFactory.sol
-rain.flare/blob/main/src/err/ErrFtso.sol
-rain.flare/blob/main/src/lib/registry/LibFlareContractRegistry.sol

The code repository contains unit tests.

Documentation was provided.

Report Update - 1

The **Rain Language Team** team has updated its code in accordance with the issues identified in the first review.

The updated code was reviewed at the following commits:
https://github.com/cyclofinance/cyclo.sol/commit/6d6ea0c8ee27b07597472d60f6cb0de26e21f695
https://github.com/gildlab/ethgild/commit/0f5efdb413f079ac6938ac6641fca2450d670c9c

Report Update - 2

The **Rain Language Team** team has updated its code.

The updated code was reviewed at the following commits:
https://github.com/cyclofinance/cyclo.sol/commit/8d1ad33044076e0cea75b6e25e285b8174cd590e
https://github.com/gildlab/ethgild/commit/eef10646cd7d2f72d634217bd3410ba82dc64da3

Report Update - 3

The **Rain Language Team** team has updated its code.

The updated code was reviewed at the following commits:
https://github.com/cyclofinance/cyclo.sol/commit/2ee2b15fa8daa81e6eb432822f570f363c111415
https://github.com/gildlab/ethgild/commit/abbbd367998a56ebb15f69a117b621d1c64a3718

Procedure

The audit includes the following procedures:

- code analysis by the Slither static analyzer, followed by manual analysis and selection of problems
- manual code analysis, including logic check
- checking the business logic for compliance with the documentation (or white paper)

In addition, during the audit, we also provide recommendations on optimizing smart contracts and using best practices.

Summary of audit findings

Severity	Count
HIGH	0
MEDIUM	0
LOW	2
INFORMATIONAL	2
TOTAL	**4**

Severity classification

High severity issues

High severity issues can lead to a direct or indirect loss of funds by both users and owners, a serious violation of the logic of the contract, including through the intervention of third parties. Such issues require immediate correction.

Medium severity issues

Medium severity issues may cause contract functionality to fail or behave incorrectly. Also, medium severity issues can cause financial damage. Such issues require increased attention.

Low severity issues

Low severity issues do not have a major security impact. It is recommended that such issues be taken into account.

Informational severity issues

These issues provide general guidelines for writing code as well as best practices.

Smart contract CycloReceipt [EGD-01]

The CycloReceipt contract is an implementation of the Receipt contract, which extends the ERC1155 standard. It includes a function uri that generates a JSON metadata string for a given token ID, encoding it in Base64 format. The metadata includes details such as the number of decimals, a description of the receipt's redeemable value, an image URI, and the name of the receipt. The image URI is a constant pointing to an SVG of the Cyclo logo stored on IPFS. The contract uses several libraries for fixed-point arithmetic and formatting.

No issues have been identified.

Smart contract Receipt [EGD-02]

The Receipt contract implements the IReceiptV1 interface and extends Ownable and ERC1155 from OpenZeppelin. It is designed to be used as a clonable implementation in a factory pattern. The contract includes functions for minting, burning, and transferring tokens, all of which can only be executed by the owner. It also emits receipt information events when tokens are minted, burned, or transferred.

No issues have been identified.

Smart contract LibFixedPointDecimalArithmeticOpenZeppelin [EGD-03]

The LibFixedPointDecimalArithmeticOpenZeppelin library provides utility functions for fixed-point arithmetic using 18 decimal places. It leverages the OpenZeppelin Math library to perform these operations with specified rounding directions.

No issues have been identified.

Smart contract LibFixedPointDecimalFormat [EGD-04]

The LibFixedPointDecimalFormat library provides a utility function to convert fixed-point decimal values to their string representation. The main function, fixedPointToDecimalString, takes a fixed-point decimal value and converts it to a string, removing trailing zeros and including decimals only if the value is non-integer.

No issues have been identified.

Smart contract FixedPointDecimalConstants [EGD-05]

The FixedPointDecimalConstants contract defines a set of constants used for fixed-point arithmetic in Solidity. These constants facilitate precise mathematical operations by adopting conventions similar to those used in Ethereum (wei) and most ERC20 tokens.

No issues have been identified.

Smart contract ERC20PriceOracleReceiptVault [EGD-06]

The ERC20PriceOracleReceiptVault contract extends the ReceiptVault contract and integrates a price oracle to determine the minting ratio for shares. It uses the price from the oracle to set the ID for receipts and to calculate the share ratio. The contract includes an initialization function to set up the price oracle and the underlying receipt vault. The `_nextId()` function fetches the current price from the oracle to use as the ID, and the `_shareRatioUserAgnostic()` function returns the ID as the share ratio. The contract emits an event upon initialization to log the configuration details.

No issues have been identified.

Smart contract SceptreStakedFlrOracle [EGD-07]

The SceptreStakedFlrOracle contract extends the PriceOracleV2 abstract contract. It overrides the `_price()` function to return the price of staked FLR using the LibSceptreStakedFlare library. The `_price()` function is marked as view since it does not modify the state.

No issues have been identified.

Smart contract TwoPriceOracleV2 [EGD-08]

The TwoPriceOracleV2 contract extends the PriceOracleV2 abstract contract. It combines two price feeds, a base and a quote, to derive a new price by dividing the base price by the quote price using fixed-point arithmetic. The contract stores the base and quote price oracles as immutable state variables and emits a Construction event upon deployment. The `_price()` function is overridden to fetch prices from the base and quote oracles, divide them, and return the result.

ID: **EGD13-01**	Severity: **Low**	Status: **Fixed**

Lack of Non-Zero and Decimal Validation for Constructor Parameters

The TwoPriceOracleV2 contract's constructor does not validate that the base and quote parameters are non-zero. Additionally, there is no validation to ensure that the oracles' decimals are consistent, which is crucial for accurate price calculations. This can lead to incorrect contract initialization and potential calculation errors if invalid parameters are provided.

Smart contract FtsoV2LTSFeedOracle [EGD-09]

The FtsoV2LTSFeedOracle contract extends the PriceOracleV2 abstract contract. It uses a feed ID and a stale duration to fetch prices from an external library LibFtsoV2LTS. The contract stores these parameters as immutable state variables and emits a Construction event upon deployment. The `_price()` function is overridden to return the price fetched using the feed ID and stale duration.

ID: **EGD13-02**	Severity: **Low**	Status: **Fixed**

Lack of Non-Zero Validation for Constructor Parameters

The FtsoV2LTSFeedOracle contract's constructor does not validate that the feedId and staleAfter parameters are non-zero. This can lead to incorrect contract initialization if invalid parameters are provided, potentially causing the contract to behave unexpectedly or fail to function as intended.

Smart contract LibFtsoV2LTS [EGD-10]

The LibFtsoV2LTS library provides functionality to fetch the value of a feed from the Flare Time Series Oracle (FTSO) using version 2 Long Term Stable (LTS). It includes a set of predefined feed IDs for various cryptocurrency/USD pairs. The main function, ftsoV2LTSGetFeed, retrieves the feed value and ensures it is not stale by checking the timestamp against a provided timeout. It uses inline assembly to handle memory allocation for the feed IDs and calculates the necessary fee for the retrieval. If the feed value is stale, it reverts the transaction with a StalePrice error.

No issues have been identified.

Smart contract PriceOracleV2 [EGD-11]

The PriceOracleV2 contract is an abstract contract that implements the IPriceOracleV2 interface. It defines a hook `_price()` for derived contracts to implement their own price logic. The `price()` function calls `_price()`, sends any balance to the caller, and returns the price. The contract also includes fallback and receive functions to accept refunds.

No issues have been identified.

Smart contract LibSceptreStakedFlare [EGD-12]

The LibSceptreStakedFlare library provides functionality related to the sFLR (Staked Flare) contract. It defines a constant `SFLR_CONTRACT` which is an immutable reference to the sFLR contract address. The library includes a function getSFLRPerFLR18 that returns the fixed 18 decimal place ratio of sFLR to FLR by calling the `getSharesByPooledFlr()` method on the sFLR contract.

No issues have been identified.

Smart contract ReceiptVault [EGD-13]

The ReceiptVault contract is an abstract implementation of an ERC4626-like vault that integrates ERC1155 receipts. It allows for the minting and burning of ERC20 shares tied to specific deposit events, represented by ERC1155 receipts. The contract includes functions for depositing, minting, withdrawing, and redeeming assets, while ensuring that each operation is tied to a specific receipt ID. It uses OpenZeppelin libraries for security and upgradeability. The contract also includes various checks and error handling to ensure the integrity of operations.

ID: **EGD13-03**	Severity: **Informational**	Status: **Fixed**

Incomplete Documentation References

The ReceiptVault contract contains several references to the `@inheritdoc IReceiptVaultV1` tag in its documentation comments. However, these references lead to non-complete documentation, which can cause confusion and misinterpretation for developers and auditors trying to understand the contract's functionality and intended behavior.

ID: **EGD13-04**	Severity: **Informational**	Status: **Fixed**

Public Functions Can Be Declared External

In the ReceiptVault contract, some functions such as `maxWithdraw()`, `previewWithdraw()`, and `withdraw()` are declared as public. However, these functions do not need to be called internally

within the contract and can be declared as external to save gas. Declaring functions as external instead of public can reduce gas costs because external functions are more efficient in terms of gas usage when called from outside the contract.

Smart contract CloneFactory [EGD-14]

The CloneFactory contract is a minimal implementation of the ICloneableFactoryV2 interface that uses OpenZeppelin's Clones library to create EIP1167 clones of a reference bytecode. It ensures the implementation address has code to avoid common mistakes and emits an event when a new clone is created. The contract also checks the return value of the initialize function on the cloned contract to ensure successful initialization, reverting the transaction if initialization fails.

No issues have been identified.

Smart contract ErrFtso [EGD-15]

The ErrFtso contract defines custom error types related to the Flare Time Series Oracle (FTSO). These errors handle various issues such as inactive FTSOs, prices not being finalized, stale prices, and inconsistent FTSO values.

No issues have been identified.

Smart contract LibFlareContractRegistry [EGD-16]

The LibFlareContractRegistry library provides utility functions to retrieve specific contract addresses from the Flare contract registry. It defines constants for the canonical names of the FTSO registry, FTSO V2 LTS, and FeeCalculator contracts. The library includes three functions: getFtsoRegistry, getFtsoV2LTS, and getFeeCalculator, which return the respective contract interfaces by querying the Flare contract registry using these canonical names. This simplifies access to these contracts within other parts of the code.

No issues have been identified.

The list of additional risks that should be considered by the development team

- The `receiptInformation()` function in the Receipt contract can be invoked at any time by any user. However, there is no specific event type associated with this function. This can lead to potential confusion when reading and interpreting events, as it may not be clear which function invocation triggered a particular event.
- The LibFtsoV2LTS library contains hardcoded feed IDs for various cryptocurrency/USD pairs. While these addresses are assumed to be correct, they may need to be updated if the project is deployed on another blockchain or if the feed addresses change. Relying on hardcoded addresses can lead to issues if the addresses are incorrect or need to be modified, as it requires changes to the source code and redeployment of the contract.
- In a few upgradeable contracts, the constructor is used to initialize immutable variables. Since the value of immutable variables is stored in the bytecode, it will be shared among all proxies pointing to a given contract instead of being stored in each proxy's storage.
- Most calculations in the project assume that tokens have 18 decimals. While this is a common standard, it is not universally true. Tokens with different decimal places may lead to incorrect calculations and potential vulnerabilities.

Conclusion

During the audit 2 low severity issues were found.

Fixing issues and covering them with tests is strongly advised.

Conclusion Update. All issues have been fixed in the updated code.

Disclaimer

Note that smart contract audit provided by Protofire is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, Protofire recommends proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Smart contract audit provided by Protofire shall not be used as investment or financial advice.