BLOCKSEC

Security Audit
Report for Rho Contracts

Date: August 23, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Rho Markets
Target	Rho Contracts

Version History

Version	Date	Description
1.0	August 23, 2024	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-09-12 08:47:30+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The focus of this audit is on Rho Contracts [1] of Rho Markets. Rho Markets is the first native lending protocol on the Scroll ecosystem, based on an overcollateralized lending model backed by the Scroll team.

Please note that Rho Contracts is based on the Compound protocol [2] and Moonwell protocol [3] which are considered as trusted codebases. Additionally, all dependencies are considered reliable in terms of both functionality and security. The security issues of the forked logic (i.e., Compound, Moonwell) are beyond the scope of the audit. This audit is focused on the code changes in files:

```
1  src/oracles/LinkedAssetAggregator.sol
2  src/oracles/PriceOracle.sol
3  src/oracles/PriceOracleV2.sol
4  src/oracles/ChainlinkOracle.sol
5  src/oracles/Api3LinkedAggregator.sol
6  src/oracles/Api3Aggregator.sol
7  src/oracles/TempChainlinkAggregator.sol
8  src/RErc20.sol
9  src/CarefulMath.sol
10 src/Exponential.sol
11 src/Unitroller.sol
12 src/ErrorReporter.sol
13 src/RToken.sol
14 src/Comptroller.sol
15 src/Rate.sol
16 src/utils/FixedPointMath.sol
17 src/utils/Addresses.sol
18 src/Timelock.sol
19 src/RErc20Delegator.sol
20 src/REther.sol
21 src/RErc20DelegatorFactory.sol
22 src/RErc20Delegate.sol
23 src/irm/WhitePaperInterestRateModel.sol
24 src/irm/InterestRateModel.sol
25 src/irm/JumpRateModelV2.sol
26 src/irm/JumpRateModel.sol
27 src/ComptrollerStorage.sol
```

[1] https://github.com/rhomarkets/Rho_Contracts.git

[2] https://github.com/compound-finance/compound-protocol

[3] https://github.com/moonwell-fi/moonwell-contracts-v2

```
28 src/rewards/MultiRewardDistributorCommon.sol
29 src/rewards/MultiRewardDistributor.sol
30 src/ExponentialNoError.sol
```

Listing 1.1: Audit Scope for this Report

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Rho Contracts	`Version 1`	`b2376aab5d737284e82d5d1345895722e41b0902`
	`Version 2`	`639ea99d4c4795e06e195708941a54dd36eef808`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Permission management
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

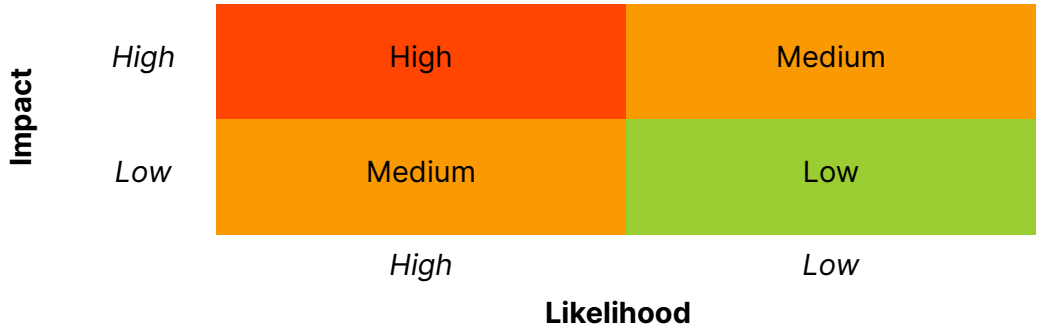
To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [4] and Common Weak-

[4] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

ness Enumeration [5]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification



Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[5] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **seventeen** potential security issues. Besides, we have **four** recommendations and **two** notes.

- High Risk: 9
- Medium Risk: 6
- Low Risk: 2
- Recommendation: 4
- Note: 2

ID	Severity	Description	Category	Status
1	Low	Potential underflow with `decimals()` exceeding 18	Software Security	Confirmed
2	Medium	Incorrect `blocksPerYear` in contract `JumpRateModel`	Software Security	Confirmed
3	Medium	Incorrectly check on `redeemAmountIn` in function `redeemFresh()`	Software Security	Fixed
4	Medium	Lack of initializing `gracePeriodTime` in contract `PriceOracleV2`	Software Security	Confirmed
5	High	Potential lock of funds due to unhandled errors	Software Security	Confirmed
6	High	Incorrect calculation on `priceInfo.expo` in contract `PriceOracleV2`	Software Security	Fixed
7	High	Shared `EmissionToken` conflict across markets	DeFi Security	Fixed
8	High	Incorrect scale calculations in contract `JumpRateModelV2`	DeFi Security	Fixed
9	High	Incorrect checks in function `liquidateBorrowAllowed()`	DeFi Security	Fixed
10	High	Incorrect rounding direction in function `redeemFresh()`	DeFi Security	Fixed
11	High	Lack of `userData` removals	DeFi Security	Fixed
12	Medium	Lack of stale price checks in oracle queries	DeFi Security	Confirmed
13	High	Potential enabling a deprecated market in the `_setProtocalPaused` contract	DeFi Security	Fixed
14	Medium	Lack of try-catch pattern in function `mintWithPermit()`	DeFi Security	Fixed
15	Medium	Potential inconsistency in the usage of the interface for the `ERC20` token	DeFi Security	Fixed
16	High	Inconsistent units of `borrowRate` and `blockDelta`	DeFi Security	Fixed

17	Low	Lack of check in function `addOracle()`	DeFi Security	Fixed
18	-	Incorrect comments and error message for function `_setBlackList()`	Recommendation	Fixed
19	-	Incorrect name for function `getBlockNumber()` of contract `Comptroller`	Recommendation	Fixed
20	-	Remove redundant codes	Recommendation	Confirmed
21	-	Deprecated function	Recommendation	Fixed
22	-	Potential centralization risks	Note	-
23	-	Potential inflation attack due to empty markets	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential underflow with `decimals()` exceeding 18

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the contract `PriceOracleV2`, the calculation of `decimalDelta` is `18 - uint256(aggregator.decimals())`. However, this calculation may revert when `decimals() > 18`. This might lead to denial of service. The contracts `Api3LinkedAggregator`, `ChainlinkOracle`, and `LinkedAssetAggregator` have the same problem.

```
129   function getChainlinkPrice(RToken rToken) public view returns (uint256, uint256, uint256) {
130       address underlyingAddr = address(getUnderlying(rToken));
131
132       address priceFeed = chainlinkPriceFeeds[underlyingAddr];
133
134       (, int256 answer, uint256 startedAt,,) = sequencerUptimeFeed.latestRoundData();
135
136       bool isSequencerUp = answer == 0;
137       if (!isSequencerUp) {
138           revert SequencerDown();
139       }
140
141       uint256 timeSinceUp = block.timestamp - startedAt;
142       if (timeSinceUp <= gracePeriodTime) {
143           revert GracePeriodNotOver();
144       }
145
146       AggregatorV3Interface aggregator = AggregatorV3Interface(priceFeed);
147       (, int256 price,, uint256 updatedAt,) = aggregator.latestRoundData();
148
```

```
149         bool isPriceFresh = (block.timestamp - updatedAt) < freshCheck;
150         if (!isPriceFresh) {
151             revert PriceNotFresh();
152         }
153
154         uint256 rawPrice = uint256(price);
155         uint256 decimals = uint256(aggregator.decimals());
156         uint256 decimalDelta = 18 - uint256(aggregator.decimals());
157
158         uint256 scaledPrice = rawPrice * (10 ** decimalDelta);
159
160         return (rawPrice, scaledPrice, decimals);
161     }
```

Listing 2.1: src/oracles/PriceOracleV2.sol

```
27  function latestRoundData()
28  public
29  view
30  override
31  returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80
        answeredInRound)
32{
33  (int224 value,) = exchangeRateFeed.read();
34
35
36  (
37      uint80 tokenRoundId,
38      int256 ethPrice,
39      uint256 tokenStartedAt,
40      uint256 tokenUpdatedAt,
41      uint80 tokenAnsweredInRound
42  ) = originPriceFeed.latestRoundData();
43
44
45  uint256 scaledExchangeRate = uint256(uint224(value));
46  uint256 scaledEthPrice = uint256(ethPrice) * 10 ** (18 - uint256(originPriceFeed.decimals()));
47
48
49  int256 finalPrice = int256((scaledEthPrice * scaledExchangeRate) / 1e18);
50
51
52  return (tokenRoundId, finalPrice, tokenStartedAt, tokenUpdatedAt, tokenAnsweredInRound);
53}
```

Listing 2.2: src/oracles/Api3LinkedAggregator.sol

```
70  function getPrice(RToken rToken) internal view returns (uint256 price) {
71      EIP20Interface token = EIP20Interface(
72          RErc20(address(rToken)).underlying()
73      );
74
75      if (prices[address(token)] != 0) {
```

```
76          price = prices[address(token)];
77      } else {
78          price = getChainlinkPrice(getFeed(token.symbol()));
79      }
80
81      uint256 decimalDelta = 18 - (uint256(token.decimals()));
82      // Ensure that we don't multiply the result by 0
83      if (decimalDelta > 0) {
84          return price * (10 ** decimalDelta);
85      } else {
86          return price;
87      }
88  }
```

Listing 2.3: src/oracles/ChainlinkOracle.sol

```
93  function getChainlinkPrice(
94      AggregatorV3Interface feed
95  ) internal view returns (uint256) {
96      (, int256 answer, , uint256 updatedAt, ) = AggregatorV3Interface(feed)
97          .latestRoundData();
98      require(answer > 0, "Chainlink price cannot be lower than 0");
99      require(updatedAt != 0, "Round is in incompleted state");
100
101     // Chainlink USD-denominated feeds store answers at 8 decimals
102     uint256 decimalDelta = 18 - (feed.decimals());
103     // Ensure that we don't multiply the result by 0
104     if (decimalDelta > 0) {
105         return uint256(answer) * (10 ** decimalDelta);
106     } else {
107         return uint256(answer);
108     }
109 }
```

Listing 2.4: src/oracles/ChainlinkOracle.sol

```
26  function latestRoundData()
27  public
28  view
29  override
30  returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80
        answeredInRound)
31{
32  (
33      uint80 tokenRoundId,
34      int256 exchangeRate,
35      uint256 tokenStartedAt,
36      uint256 tokenUpdatedAt,
37      uint80 tokenAnsweredInRound
38  ) = exchangeRateFeed.latestRoundData();
39
40  (, int256 ethPrice,,,) = originPriceFeed.latestRoundData();
41
```

```
42    uint256 scaledExchangeRate = uint256(exchangeRate) * 10 ** (18 - uint256(exchangeRateFeed.
          decimals()));
43    uint256 scaledEthPrice = uint256(ethPrice) * 10 ** (18 - uint256(originPriceFeed.decimals()));
44
45    int256 finalPrice = int256((scaledEthPrice * scaledExchangeRate) / 1e18);
46
47    return (tokenRoundId, finalPrice, tokenStartedAt, tokenUpdatedAt, tokenAnsweredInRound);
48}
```

Listing 2.5: src/oracles/LinkedAssetAggregator.sol

Impact Denial of service.

Suggestion Check whether `decimals() > 18` and add the logic when `decimals() > 18`.

Feedback from the project In practice, most mainstream oracles do not provide price data with a precision exceeding 18 decimals. If such cases arise, we will address them at the `Aggregator` level. The `ChainlinkOracle.sol` contract and related components have been removed as they are no longer in use.

2.1.2 Incorrect `blocksPerYear` in contract `JumpRateModel`

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The blocksPerYear is set as 9,556,363 in the contract JumpRateModel, which is incorrect since the block time on the Scroll chain is 3s and it should be 10,512,000.

```
19    /**
20    * @notice The approximate number of blocks per year that is assumed by the interest rate model
21    */
22  uint256 public constant blocksPerYear = 9556363;
```

Listing 2.6: src/irm/JumpRateModel.sol

Impact The APY is incorrect.

Suggestion Revise the constant.

Feedback from the project Given that the block time on the Scroll network may be affected by network conditions and may not consistently achieve the expected 4x block speed, we have adjusted the calculation to use 90% of the 4x block speed. This deviation is within our expected range.

2.1.3 Incorrectly check on `redeemAmountIn` in function `redeemFresh()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `redeemFresh` determines whether the user is redeeming by a specified amount of `rTokens` or underlying tokens by checking whether the `redeemTokensIn` or

`redeemAmountIn` is greater than zero. Additionally, when the specified amount is the `type(uint256).max`, the function is designed to automatically redeem all `rTokens` of the user. However, the condition `redeemAmountIn == type(uint256).max` on Line 667 is incorrect because the `redeemAmountIn` is expected to be zero when `redeemTokensIn > 0`.

```
645   function redeemFresh(address payable redeemer, uint256 redeemTokensIn, uint256 redeemAmountIn)
646   internal
647   returns (uint256)
648{
649   require(redeemTokensIn == 0 || redeemAmountIn == 0, "one of redeemTokensIn or redeemAmountIn
          must be zero");
650
651   RedeemLocalVars memory vars;
652
653   /* exchangeRate = invoke Exchange Rate Stored() */
654   (vars.mathErr, vars.exchangeRateMantissa) = exchangeRateStoredInternal();
655   if (vars.mathErr != MathError.NO_ERROR) {
656       return failOpaque(Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_RATE_READ_FAILED, uint256(
              vars.mathErr));
657   }
658
659   /* If redeemTokensIn > 0: */
660   if (redeemTokensIn > 0) {
661       /*
662        * We calculate the exchange rate and the amount of underlying to be redeemed:
663        *  redeemTokens = redeemTokensIn
664        *  redeemAmount = redeemTokensIn x exchangeRateCurrent
665        */
666
667       if (redeemAmountIn == type(uint256).max) {
668           vars.redeemTokens = accountTokens[redeemer];
669       } else {
670           vars.redeemTokens = redeemTokensIn;
671       }
672
673       (vars.mathErr, vars.redeemAmount) =
674           mulScalarTruncate(Exp({mantissa: vars.exchangeRateMantissa}), vars.redeemTokens);
675       if (vars.mathErr != MathError.NO_ERROR) {
676           return failOpaque(
677               Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_TOKENS_CALCULATION_FAILED, uint256(
                      vars.mathErr)
678           );
679       }
680   } else {
681       /*
682        * We get the current exchange rate and calculate the amount to be redeemed:
683        *  redeemTokens = redeemAmountIn / exchangeRate
684        *  redeemAmount = redeemAmountIn
685        */
686       if (redeemAmountIn == type(uint256).max) {
687           vars.redeemTokens = accountTokens[redeemer];
688
689           (vars.mathErr, vars.redeemAmount) =
```

```
690             mulScalarTruncate(Exp({mantissa: vars.exchangeRateMantissa}), vars.redeemTokens);
691         if (vars.mathErr != MathError.NO_ERROR) {
692             return failOpaque(
693                 Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_TOKENS_CALCULATION_FAILED, uint256
                        (vars.mathErr)
694             );
695         }
696     } else {
697         vars.redeemAmount = redeemAmountIn;
698
699         (vars.mathErr, vars.redeemTokens) =
700             divScalarByExpTruncate(redeemAmountIn, Exp({mantissa: vars.exchangeRateMantissa}));
701         if (vars.mathErr != MathError.NO_ERROR) {
702             return failOpaque(
703                 Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_AMOUNT_CALCULATION_FAILED, uint256
                        (vars.mathErr)
704             );
705         }
706     }
707 }
```

Listing 2.7: src/RToken.sol

Impact Users are unable to pass `type(uint256).max` as `redeemTokensIn` to redeem all collateral as expected.

Suggestion Revise the condition from `redeemAmountIn == type(uint256).max` to `redeemTokensIn == type(uint256).max`.

2.1.4 Lack of initializing `gracePeriodTime` in contract `PriceOracleV2`

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description Contract `PriceOracleV2` doesn't initialize the `gracePeriodTime` variable, which is used to check if the price feed is stale, in function `initialize()`. Meanwhile, there are no checks to ensure `gracePeriodTime` is bigger than zero in the function `setGracePeriodTime()`. If users call the function `getChainlinkPrice()` with an uninitialized `gracePeriodTime`, they may get stale prices.

```
43  function initialize() external initializer {
44      __Ownable_init(msg.sender);
45  }
```

Listing 2.8: src/oracles/PriceOracleV2.sol

```
60  function setGracePeriodTime(uint256 gracePeriodTime_) external onlyOwner {
61      uint256 oldGracePeriodTime = gracePeriodTime;
62      gracePeriodTime = gracePeriodTime_;
63      emit UpdateGracePeriodTime(oldGracePeriodTime, gracePeriodTime_);
64  }
```

Listing 2.9: src/oracles/PriceOracleV2.sol

```
129   function getChainlinkPrice(RToken rToken) public view returns (uint256, uint256, uint256) {
130       address underlyingAddr = address(getUnderlying(rToken));
131
132       address priceFeed = chainlinkPriceFeeds[underlyingAddr];
133
134       (, int256 answer, uint256 startedAt,,) = sequencerUptimeFeed.latestRoundData();
135
136       bool isSequencerUp = answer == 0;
137       if (!isSequencerUp) {
138           revert SequencerDown();
139       }
140
141       uint256 timeSinceUp = block.timestamp - startedAt;
142       if (timeSinceUp <= gracePeriodTime) {
143           revert GracePeriodNotOver();
144       }
145
146       AggregatorV3Interface aggregator = AggregatorV3Interface(priceFeed);
147       (, int256 price,, uint256 updatedAt,) = aggregator.latestRoundData();
148
149       bool isPriceFresh = (block.timestamp - updatedAt) < freshCheck;
150       if (!isPriceFresh) {
151           revert PriceNotFresh();
152       }
153
154       uint256 rawPrice = uint256(price);
155       uint256 decimals = uint256(aggregator.decimals());
156       uint256 decimalDelta = 18 - uint256(aggregator.decimals());
157
158       uint256 scaledPrice = rawPrice * (10 ** decimalDelta);
159
160       return (rawPrice, scaledPrice, decimals);
161   }
```

Listing 2.10: src/oracles/PriceOracleV2.sol

Impact The prices may be stale due to the uninitialized `gracePeriodTime`.

Suggestion Initialize the `gracePeriodTime` variable in the function `initialize()` and ensure it's bigger than zero in the function `setGracePeriodTime()`.

Feedback from the project It is set in the deployment script and has already been configured.

2.1.5 Potential lock of funds due to unhandled errors

Severity High

Status Confirmed

Introduced by `Version 1`

Description In the contract `REther`, when the underlying token is the native token, the function `mint()` does not correctly handle the error returned by the function `mintInternal()`. If errors occur, the transaction won't revert but return an error instead, meanwhile, the native token that has already been transferred will be not refunded, leading to users' asset loss. The functions `repayBorrow()`, `repayBorrowBehalf()`, `liquidateBorrow()`, and `receive()` have the same problem.

```
50  function mint() external payable {
51      mintInternal(msg.value);
52  }
```

Listing 2.11: src/REther.sol

```
499 function mintInternal(uint256 mintAmount) internal nonReentrant returns (uint256, uint256) {
500     uint256 error = accrueInterest();
501     if (error != uint256(Error.NO_ERROR)) {
502         // accrueInterest emits logs on errors, but we still want to log the fact that an
                attempted borrow failed
503         return (fail(Error(error), FailureInfo.MINT_ACCRUE_INTEREST_FAILED), 0);
504     }
505     // mintFresh emits the actual Mint event if successful and logs on errors, so we don't need
            to
506     return mintFresh(msg.sender, mintAmount);
507 }
```

Listing 2.12: src/RToken.sol

```
526 function mintFresh(address minter, uint256 mintAmount) internal returns (uint256, uint256) {
527     /* Fail if mint not allowed */
528     uint256 allowed = comptroller.mintAllowed(address(this), minter, mintAmount);
529     if (allowed != 0) {
530         return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.MINT_COMPTROLLER_REJECTION,
                allowed), 0);
531     }
532
533     /* Verify market's block timestamp equals current block timestamp */
534     if (accrualBlockNumber != getBlockNumber()) {
535         return (fail(Error.MARKET_NOT_FRESH, FailureInfo.MINT_FRESHNESS_CHECK), 0);
536     }
537
538     MintLocalVars memory vars;
539
540     (vars.mathErr, vars.exchangeRateMantissa) = exchangeRateStoredInternal();
541     if (vars.mathErr != MathError.NO_ERROR) {
542         return (failOpaque(Error.MATH_ERROR, FailureInfo.MINT_EXCHANGE_RATE_READ_FAILED,
                uint256(vars.mathErr)), 0);
543     }
544
545     ////////////////////////
546     // EFFECTS & INTERACTIONS
547     // (No safe failures beyond this point)
548
549     /*
```

```
550         * We call `doTransferIn` for the minter and the mintAmount.
551         * Note: The rToken must handle variations between ERC-20 and GLMR underlying.
552         * `doTransferIn` reverts if anything goes wrong, since we can't be sure if
553         * side-effects occurred. The function returns the amount actually transferred,
554         * in case of a fee. On success, the rToken holds an additional `actualMintAmount`
555         * of cash.
556         */
557        vars.actualMintAmount = doTransferIn(minter, mintAmount);
558
559        /*
560         * We get the current exchange rate and calculate the number of rTokens to be minted:
561         * mintTokens = actualMintAmount / exchangeRate
562         */
563
564        (vars.mathErr, vars.mintTokens) =
565            divScalarByExpTruncate(vars.actualMintAmount, Exp({mantissa: vars.exchangeRateMantissa
                }));
566        require(vars.mathErr == MathError.NO_ERROR, "MINT_EXCHANGE_CALCULATION_FAILED");
567
568        /*
569         * We calculate the new total supply of rTokens and minter token balance, checking for
              overflow:
570         * totalSupplyNew = totalSupply + mintTokens
571         * accountTokensNew = accountTokens[minter] + mintTokens
572         */
573        (vars.mathErr, vars.totalSupplyNew) = addUInt(totalSupply, vars.mintTokens);
574        require(vars.mathErr == MathError.NO_ERROR, "MINT_NEW_TOTAL_SUPPLY_CALCULATION_FAILED");
575
576        (vars.mathErr, vars.accountTokensNew) = addUInt(accountTokens[minter], vars.mintTokens);
577        require(vars.mathErr == MathError.NO_ERROR, "MINT_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED");
578
579        /* We write previously calculated values into storage */
580        totalSupply = vars.totalSupplyNew;
581        accountTokens[minter] = vars.accountTokensNew;
582
583        /* We emit a Mint event, and a Transfer event */
584        emit Mint(minter, vars.actualMintAmount, vars.mintTokens);
585        emit Transfer(address(this), minter, vars.mintTokens);
586
587        /* We call the defense hook */
588        // unused function
589
590        comptroller.enterAllMarkets(minter);
591
592        return (uint256(Error.NO_ERROR), vars.actualMintAmount);
593    }
```

Listing 2.13: src/RToken.sol

Impact Users' assets may be locked.

Suggestion Check the return value and revert if it does not equal with `uint256(Error.NO_ERROR)` in functions `repayBorrow()`, `repayBorrowBehalf()`, `liquidateBorrow()`, and `receive()` in the

contract `REther`.

Feedback from the project This is a known issue, and we will not make changes to `REther.sol`. In the future, this asset contract will be deprecated and replaced with a new, upgraded contract.

2.1.6 Incorrect calculation on `priceInfo.expo` in contract `PriceOracleV2`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The functions `getStandardizedPrice()` and `getPythPrice()` use the `expo` returned from `getPythPrice()` as a positive value. However, `expo` is a negative value in most cases, which could lead to DoS and other unexpected results.

```
263   function getStandardizedPrice(RToken rToken, bytes4 functionSig) internal view returns (
          uint256 price) {
264       if (PriceOracleV2(this).getChainlinkPrice.selector == functionSig) {
265           (, uint256 price_,) = getChainlinkPrice(rToken);
266           price = price_;
267       } else if (PriceOracleV2(this).getPythPrice.selector == functionSig) {
268           (int64 pythPrice, int32 expo) = getPythPrice(rToken);
269           price = uint256(uint64(pythPrice)) * 10 ** uint256(uint32(18 - expo));
270       }
271   }
272
273   function getUnderlyingScaledPrice(RToken rToken, bytes4 functionSig) internal view returns (
          uint256 price) {
274       ERC20 underlying = getUnderlying(rToken);
275       uint256 decimals = address(underlying) == address(0) ? 18 : underlying.decimals();
276
277       uint256 feedDecimals;
278
279       if (functionSig == PriceOracleV2(this).getChainlinkPrice.selector) {
280           (uint256 rawPrice,, uint256 decimals_) = getChainlinkPrice(rToken);
281           feedDecimals = decimals_;
282           price = scalePrice(rawPrice, feedDecimals, decimals);
283       } else if (functionSig == PriceOracleV2(this).getPythPrice.selector) {
284           (int64 pythPrice, int32 expo) = getPythPrice(rToken);
285           feedDecimals = uint256(uint32(expo));
286           price = uint256(uint64(pythPrice));
287           price = scalePrice(price, feedDecimals, decimals);
288       }
289
290       // Multiply by 10^36 and then divide by the square of the underlying token's decimals
291       price = price * 10 ** (36 - 2 * decimals);
292   }
```

Listing 2.14: src/oracles/PriceOracleV2.sol

Impact Potential DoS and other unexpected results.

Suggestion Revise the function to correctly handle the negative value of `expo`.

Note The project fixes this issue by deleting the contracts.

2.2 DeFi Security

2.2.1 Shared `EmissionToken` conflict across markets

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Due to different markets potentially being associated with the same emission tokens, the key used to access `userData` in the assets mapping may not be unique. This overlap means that accessing or modifying `userData` via the `emissionToken` key affects multiple markets simultaneously. As a result, disbursing rewards based on this shared key can inadvertently impact a user's rewards across different markets, potentially leading to asset loss or denial of service (DoS) due to incorrect or unintended data manipulation.

```
849   function disburseSupplierRewardsInternal(RToken _rToken, address _supplier, bool _sendTokens)
          internal {
850       address _rTokenAddr = address(_rToken);
851       MarketConfig[] storage configs = assets[_rTokenAddr].marketConfigs;
852
853       uint256 supplierTokens = _rToken.balanceOf(_supplier);
854
855       // Iterate over all market configs and update their indexes + timestamps
856       for (uint256 index = 0; index < configs.length; index++) {
857           MarketConfig storage _marketConfig = configs[index];
858
859           UserData storage userData = assets[_marketConfig.emissionToken].userData[_supplier];
860
861           uint256 totalRewardsOwed = calculateSupplyRewardsForUser(
862               _marketConfig.emissionToken, _marketConfig.supplyGlobalIndex, supplierTokens,
                      _supplier
863           );
864
865           // Update user's index to match global index
866           userData.supplierIndex = _marketConfig.supplyGlobalIndex;
867           // Update the user's total rewards owed
868           userData.supplierRewardsAccrued = totalRewardsOwed;
869
870           emit DisbursedSupplierRewards(
871               _rToken, _supplier, _marketConfig.emissionToken, userData.supplierRewardsAccrued
872           );
873
874           // SendRewards will attempt to send only if it has enough emission tokens to do so,
875           // and if it doesn't have enough it emits a InsufficientTokensToEmit event and returns
876           // the rewards that couldn't be sent, which are the total of what a user is owed, so we
877           // store it in supplierRewardsAccrued to make sure we don't lose rewards accrual if
                  there's
```

```
878             // not enough funds in the rewarder
879             if (_sendTokens) {
880                 // Emit rewards for this token/pair
881                 uint256 unsendableRewards =
882                     sendReward(payable(_supplier), userData.supplierRewardsAccrued, _marketConfig.
                        emissionToken);
883
884                 userData.supplierRewardsAccrued = unsendableRewards;
885             }
886         }
887     }
```

Listing 2.15: src/rewards/MultiRewardDistributor.sol

Impact `userData` may not be correctly updated.

Suggestion Use unique keys for the assets to access `userData`.

Feedback from the project We are not currently using the `MultiRewardDistributor.sol` contract for incentive distribution. This part of the code will be removed, while retaining the related interfaces for potential future upgrades.

Note The project fixes this issue by deleting the contracts.

2.2.2 Incorrect scale calculations in contract `JumpRateModelV2`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In `JumpRateModelV2` contract, the variables `baseRatePerTimestamp`, `multiplierPerTimestamp`, and `jumpMultiplierPerTimestamp` are initialized in the constructor using the parameters `baseRatePerYear`, `multiplierPerYear`, and `jumpMultiplierPerYear`. These parameters are already scaled with `1e18`, but they are scaled again during initialization, leading to incorrect results. The contract `WhitePaperInterestRateModel` has the same problem.

```
40  /**
41   * @notice Construct an interest rate model
42   * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
43   * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1
         e18)
44   * @param jumpMultiplierPerYear The multiplierPerTimestamp after hitting a specified
         utilization point
45   * @param kink_ The utilization point at which the jump multiplier is applied
46   */
47  constructor(uint256 baseRatePerYear, uint256 multiplierPerYear, uint256 jumpMultiplierPerYear,
        uint256 kink_) {
48      baseRatePerTimestamp = (baseRatePerYear * 1e18) / timestampsPerYear;
49      multiplierPerTimestamp = (multiplierPerYear * 1e18) / timestampsPerYear;
50      jumpMultiplierPerTimestamp = (jumpMultiplierPerYear * 1e18) / timestampsPerYear;
51      kink = kink_;
52
```

```
53      emit NewInterestParams(baseRatePerTimestamp, multiplierPerTimestamp,
            jumpMultiplierPerTimestamp, kink);
54  }
```

Listing 2.16: src/irm/JumpRateModelV2.sol

```
40  /**
41   * @notice Construct an interest rate model
42   * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
43   * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1
         e18)
44   */
45  constructor(uint256 baseRatePerYear, uint256 multiplierPerYear) {
46      baseRatePerTimestamp = (baseRatePerYear * 1e18) / timestampsPerYear;
47      multiplierPerTimestamp = (multiplierPerYear * 1e18) / timestampsPerYear;
48
49      emit NewInterestParams(baseRatePerTimestamp, multiplierPerTimestamp);
50  }
```

Listing 2.17: src/irm/WhitePaperInterestRateModel.sol

Impact Incorrectly scaled variables may cause inconsistencies across different interest rate models.

Suggestion Revise the calculation for the initialization of these variables to ensure accurate scaling.

2.2.3 Incorrect checks in function `liquidateBorrowAllowed()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `Comptroller` contract, the function `liquidateBorrowAllowed()` contains incorrect liquidation checks. Despite the code comments indicating that only the whitelisted addresses can liquidate when `liquidatable` is true, the current logic `(liquidatorWhiteList[liquidator] && !liquidatable)` allows non-whitelist addresses to do liquidations.

```
450  function liquidateBorrowAllowed(
451      address rTokenBorrowed,
452      address rTokenCollateral,
453      address liquidator,
454      address borrower,
455      uint256 repayAmount
456  ) external view override returns (uint256) {
457      if (protocalProtectedAccount[borrower]) {
458          return uint256(Error.UNAUTHORIZED);
459      }
460
461      if (liquidatorWhiteList[liquidator] && !liquidatable) {
462          return uint256(Error.UNAUTHORIZED);
463      }
```

```
464
465         if (!markets[rTokenBorrowed].isListed || !markets[rTokenCollateral].isListed) {
466             return uint256(Error.MARKET_NOT_LISTED);
467         }
468
469         if (!markets[rTokenCollateral].accountMembership[borrower]) {
470             return uint256(Error.MARKET_NOT_ENTERED);
471         }
472
473         uint256 borrowBalance = RToken(rTokenBorrowed).borrowBalanceStored(borrower);
474
475         /* allow accounts to be liquidated if the market is deprecated */
476         if (isDeprecated(RToken(rTokenBorrowed))) {
477             require(borrowBalance >= repayAmount, "Can not repay more than the total borrow");
478         } else {
479             /* The borrower must have shortfall in order to be liquidatable */
480             (Error err,, uint256 shortfall) = getAccountLiquidityInternal(borrower);
481
482             if (err != Error.NO_ERROR) {
483                 return uint256(err);
484             }
485
486             if (shortfall == 0) {
487                 return uint256(Error.INSUFFICIENT_SHORTFALL);
488             }
489
490             /* The liquidator may not repay more than what is allowed by the closeFactor */
491             uint256 maxClose = mul_ScalarTruncate(Exp({mantissa: closeFactorMantissa}),
                    borrowBalance);
492
493             if (repayAmount > maxClose) {
494                 return uint256(Error.TOO_MUCH_REPAY);
495             }
496         }
497         return uint256(Error.NO_ERROR);
498     }
```

Listing 2.18: src/Comptroller.sol

Impact Non-whitelist accounts can bypass the check and do liquidations.

Suggestion Use `if (liquidatable && !liquidatorWhiteList[liquidator])` instead.

2.2.4 Incorrect rounding direction in function `redeemFresh()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Currently, the rounding direction for calculating required `redeemTokens` in the function `redeemFresh()` is rounding down and in favor of the redeemers. Attackers can leverage this with an inflated exchange rate and withdraw more assets than they should, leading to asset loss.

```
681   /*
682   * We get the current exchange rate and calculate the amount to be redeemed:
683   *  redeemTokens = redeemAmountIn / exchangeRate
684   *  redeemAmount = redeemAmountIn
685   */
686  if (redeemAmountIn == type(uint256).max) {
687      vars.redeemTokens = accountTokens[redeemer];
688
689      (vars.mathErr, vars.redeemAmount) =
690          mulScalarTruncate(Exp({mantissa: vars.exchangeRateMantissa}), vars.redeemTokens);
691      if (vars.mathErr != MathError.NO_ERROR) {
692          return failOpaque(
693              Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_TOKENS_CALCULATION_FAILED, uint256(
                      vars.mathErr)
694          );
695      }
696  } else {
697      vars.redeemAmount = redeemAmountIn;
698
699      (vars.mathErr, vars.redeemTokens) =
700          divScalarByExpTruncate(redeemAmountIn, Exp({mantissa: vars.exchangeRateMantissa}));
701      if (vars.mathErr != MathError.NO_ERROR) {
702          return failOpaque(
703              Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_AMOUNT_CALCULATION_FAILED, uint256(
                      vars.mathErr)
704          );
705      }
706  }
```

Listing 2.19: src/RToken.sol

Impact Assets loss for the protocol and users.

Suggestion Calculate the `vars.redeemAmount` by multiplying the rounded-down `vars.redeemTokens` with the exchange rate.

2.2.5 Lack of `userData` removals

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `_removeEmissionConfig` removes all `MarketConfig` records associated with a specific `emissionToken` but lacks removal of the corresponding `userData` records. Additionally, the function `_removeMarket()` attempts to delete all data related to a specified asset using `delete assets[address(rToken)]`. However, in Solidity, deleting a struct can not delete the mappings within it. As a result, these residual `userData` records may be reused if a previously removed `MarketConfig` is reinstated, resulting in unexpected results.

```
404   function _removeEmissionConfig(RToken _rToken, address _emissionToken) external
          onlyComptrollersAdmin {
```

```
405         MarketConfig[] storage configs = assets[address(_rToken)].marketConfigs;
406
407         // Find the index of the config to remove
408         bool found = false;
409         uint256 removeIndex;
410         for (uint256 index = 0; index < configs.length; index++) {
411             if (configs[index].emissionToken == _emissionToken) {
412                 removeIndex = index;
413                 found = true;
414                 break;
415             }
416         }
417
418         require(found, "Emission token not found!");
419
420         // Remove the config by swapping it with the last element and popping the array
421         if (removeIndex < configs.length - 1) {
422             configs[removeIndex] = configs[configs.length - 1];
423         }
424         configs.pop();
425
426         emit ConfigRemoved(address(_rToken), _emissionToken);
427     }
```

Listing 2.20: src/rewards/MultiRewardDistributor.sol

```
434     function _removeMarket(RToken _rToken) external onlyComptrollersAdmin {
435         delete assets[address(_rToken)];
436     }
```

Listing 2.21: src/rewards/MultiRewardDistributor.sol

```
32     struct AssetData {
33         MarketConfig[] marketConfigs;
34         mapping(address => UserData) userData;
35     }
```

Listing 2.22: src/rewards/MultiRewardDistributorCommon.sol

Impact When a `MarketConfig` or asset removed is re-added, the unremoved `userData` may cause potential inconsistencies.

Suggestion Revise these functions to explicitly delete `userData` alongside the `MarketConfig` records.

Note The project fixes this issue by deleting the contracts.

2.2.6 Lack of stale price checks in oracle queries

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The contract `ChainlinkOracle` fetches prices from the specified oracles. However, it doesn't verify whether the fetched price is a stale value. The contracts `Api3Aggregator`, `Api3LinkedAggregator` and `LinkedAssetAggregator` have the same problem.

```
93  function getChainlinkPrice(
94      AggregatorV3Interface feed
95  ) internal view returns (uint256) {
96      (, int256 answer, , uint256 updatedAt, ) = AggregatorV3Interface(feed)
97          .latestRoundData();
98      require(answer > 0, "Chainlink price cannot be lower than 0");
99      require(updatedAt != 0, "Round is in incompleted state");
100
101     // Chainlink USD-denominated feeds store answers at 8 decimals
102     uint256 decimalDelta = 18 - (feed.decimals());
103     // Ensure that we don't multiply the result by 0
104     if (decimalDelta > 0) {
105         return uint256(answer) * (10 ** decimalDelta);
106     } else {
107         return uint256(answer);
108     }
109 }
```

Listing 2.23: src/oracles/ChainlinkOracle.sol

```
24  function latestRoundData()
25      public
26      view
27      override
28      returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80
            answeredInRound)
29  {
30      (int224 value,) = originPriceFeed.read();
31
32      int256 scaledTokenPrice = int256(int224(uint224(value)));
33      return (uint80(1), scaledTokenPrice, block.timestamp - 1000, block.timestamp, uint80(0));
34  }
```

Listing 2.24: src/oracles/Api3Aggregator.sol

```
27  function latestRoundData()
28      public
29      view
30      override
31      returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80
            answeredInRound)
32  {
33      (int224 value,) = exchangeRateFeed.read();
34
35      (
36          uint80 tokenRoundId,
37          int256 ethPrice,
38          uint256 tokenStartedAt,
39          uint256 tokenUpdatedAt,
```

```
40          uint80 tokenAnsweredInRound
41      ) = originPriceFeed.latestRoundData();
42
43      uint256 scaledExchangeRate = uint256(uint224(value));
44      uint256 scaledEthPrice = uint256(ethPrice) * 10 ** (18 - uint256(originPriceFeed.decimals()
            ));
45
46      int256 finalPrice = int256((scaledEthPrice * scaledExchangeRate) / 1e18);
47
48      return (tokenRoundId, finalPrice, tokenStartedAt, tokenUpdatedAt, tokenAnsweredInRound);
49  }
```

Listing 2.25: src/oracles/Api3LinkedAggregator.sol

```
26  function latestRoundData()
27      public
28      view
29      override
30      returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80
            answeredInRound)
31  {
32      (
33          uint80 tokenRoundId,
34          int256 exchangeRate,
35          uint256 tokenStartedAt,
36          uint256 tokenUpdatedAt,
37          uint80 tokenAnsweredInRound
38      ) = exchangeRateFeed.latestRoundData();
39
40      (, int256 ethPrice,,,) = originPriceFeed.latestRoundData();
41
42      uint256 scaledExchangeRate = uint256(exchangeRate) * 10 ** (18 - uint256(exchangeRateFeed.
            decimals()));
43      uint256 scaledEthPrice = uint256(ethPrice) * 10 ** (18 - uint256(originPriceFeed.decimals()
            ));
44
45      int256 finalPrice = int256((scaledEthPrice * scaledExchangeRate) / 1e18);
46
47      ...
48      return (tokenRoundId, finalPrice, tokenStartedAt, tokenUpdatedAt, tokenAnsweredInRound);
49  }
```

Listing 2.26: src/oracles/LinkedAssetAggregator.sol

Impact Stale prices may be used, leading to incorrect calculation and potentially loss of funds.

Suggestion Check the freshness of the price data.

Feedback from the project The `Api3LinkedAggregator` currently relies on `API3` as the data source for price updates. However, it does not return the timestamp of the price validation. In scenarios where temporary assets need to be launched and `Chainlink` does not yet provide the relevant price feeds, we use `API3` as an interim price data source. This is an internally acknowledged issue that does not require immediate fixing.

The `LinkedAssetAggregator` primarily relies on the exchange rate between the target token and its corresponding token to calculate prices. The exchange rate does not update as frequently as the target token's price. Without a price validation mechanism (which would require comparing historical prices, a process that is costly in practice), we do not verify exchange rate changes directly. Instead, we use offline monitoring to manage these dependent data sources.

2.2.7 Potential enabling a deprecated market in the `_setProtocalPaused` contract

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `_setProtocalPaused` is used to pause or unpause all markets. However, this could inadvertently lead to deprecated markets being unpaused when the admin passes the `state` with `true` for recovery.

```
1261  function _setProtocalPaused(bool state) public returns (bool) {
1262      require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin
          can pause");
1263      require(msg.sender == admin || state == true, "only admin can unpause");
1264
1265      for (uint256 i = 0; i < allMarkets.length; i++) {
1266          RToken rToken_ = allMarkets[i];
1267          address rToken = address(rToken_);
1268          borrowGuardianPaused[rToken] = state;
1269          mintGuardianPaused[rToken] = state;
1270          redeemGuardianPaused[rToken] = state;
1271
1272          emit ActionPaused(rToken_, "Mint", state);
1273          emit ActionPaused(rToken_, "Borrow", state);
1274          emit ActionPaused(rToken_, "Redeem", state);
1275      }
1276
1277      transferGuardianPaused = true;
1278      seizeGuardianPaused = true;
1279      return true;
1280  }
```

Listing 2.27: src/Comptroller.sol

Impact Deprecated markets can be unpaused unexpectedly.

Suggestion Revise the function to allow pausing specific markets instead of applying to all markets.

2.2.8 Lack of try-catch pattern in function `mintWithPermit()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `mintWithPermit` directly calls `token.permit()`. However permit signatures can be submitted by anyone. A malicious actor could exploit this by front-running the `mintWithPermit` invocation and submitting users' permit signatures directly, which increases the nonce recorded in the token contract. This could cause the `mintWithPermit` invocation to revert, resulting in a potential DoS attack.

```
66  function mintWithPermit(uint256 mintAmount, uint256 deadline, uint8 v, bytes32 r, bytes32 s)
67      external
68      override
69      returns (uint256)
70  {
71      IERC20Permit token = IERC20Permit(underlying);
72
73      // Go submit our pre-approval signature data to the underlying token, but
74      // explicitly fail if there is an issue.
75      token.permit(msg.sender, address(this), mintAmount, deadline, v, r, s);
76
77      (uint256 err,) = mintInternal(mintAmount);
78      return err;
79  }
```

Listing 2.28: src/RErc20.sol

Impact Potential DoS attack.

Suggestion Use the try-catch pattern to handle permit failures for mitigation.

2.2.9 Potential inconsistency in the usage of the interface for the `ERC20` token

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `Comptroller`, function `_rescueFunds()` invokes `token.transfer()`, assuming the token is an `ERC20` compatible token. However, if the token is not `ERC20` compatible and doesn't return a bool value when the function `transfer()` is invoked, the invocation will revert and the admin can't sweep the token out.

```
1062  function _rescueFunds(address _tokenAddress, uint256 _amount) external {
1063      require(msg.sender == admin, "Unauthorized");
1064
1065      IERC20 token = IERC20(_tokenAddress);
1066      // Similar to rTokens, if this is uint.max that means "transfer everything"
1067      if (_amount == type(uint256).max) {
1068          token.transfer(admin, token.balanceOf(address(this)));
1069      } else {
1070          token.transfer(admin, _amount);
1071      }
1072  }
```

Listing 2.29: src/Comptroller.sol

Impact The incompatible tokens may fail to be recovered.

Suggestion Use function `safeTransfer()` from the OpenZeppelin library.

2.2.10 Inconsistent units of `borrowRate` and `blockDelta`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `RToken`, the calculation of `simpleInterestFactor` is `mulScalar(Exp(mantissa: borrowRateMantissa), blockDelta)`. This calculation is incorrect when the `interestRateModel` is `JumpRateModelV2.sol` or `WhitePaperInterestRateModel.sol`, because the `blockDelta` is actually the delta of `block.number` while `borrowRateMantissa` is the borrowing rate per timestamp. They have inconsistent units. This will lead to less of the result of `accrueInterest()`.

```
77  function getBorrowRate(uint256 cash, uint256 borrows, uint256 reserves) public view override
        returns (uint256) {
78      uint256 util = utilizationRate(cash, borrows, reserves);
79
80      if (util <= kink) {
81          return (util * multiplierPerTimestamp) / 1e18 + baseRatePerTimestamp;
82      } else {
83          uint256 normalRate = (kink * multiplierPerTimestamp) / 1e18 + baseRatePerTimestamp;
84          uint256 excessUtil = util - kink;
85          return (excessUtil * jumpMultiplierPerTimestamp) / 1e18 + normalRate;
86      }
87  }
```

Listing 2.30: src/irm/JumpRateModelV2.sol

```
63  function getBorrowRate(uint256 cash, uint256 borrows, uint256 reserves) public view override
        returns (uint256) {
64      uint256 ur = utilizationRate(cash, borrows, reserves);
65      return (ur * multiplierPerTimestamp) / 1e18 + baseRatePerTimestamp;
66  }
```

Listing 2.31: src/irm/WhitePaperInterestRateModel.sol

```
400 function accrueInterest() public virtual override returns (uint256) {
401     /* Remember the initial block timestamp */
402     uint256 currentBlockNumber = getBlockNumber();
403     uint256 accrualBlockNumberPrior = accrualBlockNumber;
404
405     /* Short-circuit accumulating 0 interest */
406     if (accrualBlockNumberPrior == currentBlockNumber) {
407         return uint256(Error.NO_ERROR);
408     }
409
410     /* Read the previous values out of storage */
411     uint256 cashPrior = getCashPrior();
412     uint256 borrowsPrior = totalBorrows;
413     uint256 reservesPrior = totalReserves;
```

```
414         uint256 borrowIndexPrior = borrowIndex;
415
416         /* Calculate the current borrow interest rate */
417         uint256 borrowRateMantissa = interestRateModel.getBorrowRate(cashPrior, borrowsPrior,
                reservesPrior);
418         require(borrowRateMantissa <= borrowRateMaxMantissa, "borrow rate is absurdly high");
419
420         /* Calculate the number of blocks elapsed since the last accrual */
421         (MathError mathErr, uint256 blockDelta) = subUInt(currentBlockNumber,
                accrualBlockNumberPrior);
422         require(mathErr == MathError.NO_ERROR, "could not calculate block delta");
423
424         /*
425          * Calculate the interest accumulated into borrows and reserves and the new index:
426          *  simpleInterestFactor = borrowRate * blockDelta
427          *  interestAccumulated = simpleInterestFactor * totalBorrows
428          *  totalBorrowsNew = interestAccumulated + totalBorrows
429          *  totalReservesNew = interestAccumulated * reserveFactor + totalReserves
430          *  borrowIndexNew = simpleInterestFactor * borrowIndex + borrowIndex
431          */
432
433         Exp memory simpleInterestFactor;
434         uint256 interestAccumulated;
435         uint256 totalBorrowsNew;
436         uint256 totalReservesNew;
437         uint256 borrowIndexNew;
438
439         (mathErr, simpleInterestFactor) = mulScalar(Exp({mantissa: borrowRateMantissa}), blockDelta
                );
440         if (mathErr != MathError.NO_ERROR) {
441             return failOpaque(
442                 Error.MATH_ERROR,
443                 FailureInfo.ACCRUE_INTEREST_SIMPLE_INTEREST_FACTOR_CALCULATION_FAILED,
444                 uint256(mathErr)
445             );
446         }
447
448         (mathErr, interestAccumulated) = mulScalarTruncate(simpleInterestFactor, borrowsPrior);
449         if (mathErr != MathError.NO_ERROR) {
450             return failOpaque(
451                 Error.MATH_ERROR, FailureInfo.
                        ACCRUE_INTEREST_ACCUMULATED_INTEREST_CALCULATION_FAILED, uint256(mathErr)
452             );
453         }
454
455         (mathErr, totalBorrowsNew) = addUInt(interestAccumulated, borrowsPrior);
456         if (mathErr != MathError.NO_ERROR) {
457             return failOpaque(
458                 Error.MATH_ERROR, FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_BORROWS_CALCULATION_FAILED,
                        uint256(mathErr)
459             );
460         }
461
```

```
462         (mathErr, totalReservesNew) =
463             mulScalarTruncateAddUInt(Exp({mantissa: reserveFactorMantissa}), interestAccumulated,
                    reservesPrior);
464         if (mathErr != MathError.NO_ERROR) {
465             return failOpaque(
466                 Error.MATH_ERROR, FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCULATION_FAILED
                        , uint256(mathErr)
467             );
468         }
469
470         (mathErr, borrowIndexNew) = mulScalarTruncateAddUInt(simpleInterestFactor, borrowIndexPrior
                , borrowIndexPrior);
471         if (mathErr != MathError.NO_ERROR) {
472             return failOpaque(
473                 Error.MATH_ERROR, FailureInfo.ACCRUE_INTEREST_NEW_BORROW_INDEX_CALCULATION_FAILED,
                        uint256(mathErr)
474             );
475         }
476
477         ////////////////////////
478         // EFFECTS & INTERACTIONS
479         // (No safe failures beyond this point)
480
481         /* We write the previously calculated values into storage */
482         accrualBlockNumber = currentBlockNumber;
483         borrowIndex = borrowIndexNew;
484         totalBorrows = totalBorrowsNew;
485         totalReserves = totalReservesNew;
486
487         /* We emit an AccrueInterest event */
488         emit AccrueInterest(cashPrior, interestAccumulated, borrowIndexNew, totalBorrowsNew);
489
490         return uint256(Error.NO_ERROR);
491     }
```

Listing 2.32: src/RToken.sol

Impact Less of the result of `accrueInterest()`.

Suggestion Currently, the unit of `borrowRate` in the contract `JumpRateModel` is `block.number`, while in the contract `JumpRateModelV2` and `WhitePaperInterestRateModel` the unit is `block.timestamp`. Add logic to distinguish when using different interestRateModels and revise the calculation of `blockDelta` to be consistent with the different units.

Feedback from the Project The `JumpRateModel` will not be used.

2.2.11 Lack of check in function `addOracle()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `addOracle` in the `PriceOracleV2` contract does not verify that the `functionSelector` is either `getChainlinkPrice.selector` or `getPythPrice.selector`. If a non-existent selector is accidentally added, the price returned by `PriceOracleV2` could be incorrect. For example, the function `getStandardizedPrice` returns 0 when the `functionSig` is invalid.

```
180   function addOracle(RToken rToken, bytes4 functionSelector, uint256 priority) public onlyOwner
          {
181       address underlyingAddr = address(getUnderlying(rToken));
182
183       assetConfigs[underlyingAddr].push(OracleData(priority, functionSelector));
184       emit OracleAdded(underlyingAddr, functionSelector, priority);
185   }
```

Listing 2.33: src/oracles/PriceOracleV2.sol

```
263   function getStandardizedPrice(RToken rToken, bytes4 functionSig) internal view returns (
          uint256 price) {
264       if (PriceOracleV2(this).getChainlinkPrice.selector == functionSig) {
265           (, uint256 price_,) = getChainlinkPrice(rToken);
266           price = price_;
267       } else if (PriceOracleV2(this).getPythPrice.selector == functionSig) {
268           (int64 pythPrice, int32 expo) = getPythPrice(rToken);
269           price = uint256(uint64(pythPrice)) * 10 ** uint256(uint32(18 - expo));
270       }
271   }
```

Listing 2.34: src/oracles/PriceOracleV2.sol

Impact Invalid `functionSignature` configured could result in incorrect prices being returned from `PriceOracleV2`.

Suggestion Add validation for the `functionSignature` in the `addOracle` function, or implement a default revert behavior when a non-existent `functionSig` is used.

2.3 Additional Recommendation

2.3.1 Incorrect comments and error message for function `_setBlackList()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The comments and error messages for the function `_setBlackList()` are incorrect.

```
1248   /**
1249   * @notice set asset redemption paused
1250   * @dev state of account is add blacklist flag
1251   */
1252  function _setBlackList(address account_, bool state) public {
1253      require(msg.sender == admin, "only admin can unpause");
1254      blackList[account_] = state;
1255  }
```

Listing 2.35: src/Comptroller.sol

Suggestion The comment should be "set blacklist" and the error message should be "only admin can set the blacklist."

2.3.2 Incorrect name for function `getBlockNumber()` of contract `Comptroller`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The name of the function `getBlockNumber()` is incorrect since it returns the `block.timestamp`.

```
1301    function getBlockNumber() public view returns (uint256) {
1302        return block.timestamp;
1303    }
```

Listing 2.36: src/Comptroller.sol

Suggestion Rename the function to `getBlockTimestamp()`.

2.3.3 Remove redundant codes

Status Confirmed

Introduced by `Version 1`

Description Redundant code exists in the contracts and can be safely removed. For example, the constants `minRedemptionCashRequire` and `redemptionReserveFactorMaxMantissa` are declared in the `RTokenStorage` contract but are not being used.

Suggestion Remove the redundant codes.

2.3.4 Deprecated function

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `getPrice` function of pyth is deprecated, please consider using `getPriceNoOlderThan` instead.

```
89    // Public view functions to fetch price feed data
90    function getPythPrice(RToken rToken) public view override returns (int64, int32) {
91        address underlyingAddr = address(getUnderlying(rToken));
92
93
94        bytes32 priceFeedID = pythPriceFeedIDs[underlyingAddr];
95        IPythFeed priceFeed = IPythFeed(pythOracle);
96
97
98        IPythFeed.Price memory priceInfo = priceFeed.getPrice(priceFeedID);
99
```

```
100
101         if (priceInfo.price < 0) {
102             revert NegativeOraclePrice();
103         }
104
105
106         return (priceInfo.price, priceInfo.expo);
107     }
```

Listing 2.37: src/oracles/PriceOracleV2.sol

Suggestion Use function `getPriceNoOlderThan()`.

Note The project fixes this issue by deleting the contracts.

2.4 Note

2.4.1 Potential centralization risks

Introduced by `Version 1`

Description There are several important functions like `_rescueFunds()`, `setChainlinkPriceFeed()`, `_setSupplyCapGuardian()`, `updateLiquidateWhiteList()`, etc., which are only callable by the owner or admin. If their private keys are lost or compromised, it could lead to losses for the protocol and users.

2.4.2 Potential inflation attack due to empty markets

Introduced by `Version 1`

Description The first supplier to an empty market can mint X `wei` of `rToken` and then redeem X-1 `wei` of `rToken`. At this point, the market would contain only 1 `wei` of `rToken`. If other suppliers mint `rToken` afterwards, the attacker can donate `underlyingToken` to manipulate the `exchangeRate` just before others mint, resulting in others receiving 0 `wei` of `rToken`. The protocol should mint some `rToken` for newly created markets to avoid empty markets.

Feedback from the project Initial supply of `rToken` will be added in deploy scripts.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS