Rho Markets

Smart Contract Security Assessment

May 17, 2024

ABSTRACT

Dedaub was commissioned to perform a security audit of the Rho Markets protocol, to be deployed on the Scroll blockchain. The protocol is a clone of Compound v2 and, more closely, of the Moonwell protocol, modifying the original Compound code. This is a delta audit, with the underlying Moonwell protocol considered trusted. Although the team has put considerable effort into testing, we found a few minor copy-paste or search-and-replace errors in code that was apparently not thoroughly tested at the time. Additionally, we warn of standard attack vectors relative to empty markets and governance.

BACKGROUND

Dedaub performed a security audit of the Rho Markets protocol, to be deployed on the Scroll blockchain. The protocol is based on the Compound v2 codebase, also adopting the changes put in place by the Moonwell protocol, on Base and other chains. The Moonwell protocol is considered a trusted basis for the purposes of this audit. Moonwell modernizes several aspects of the Compound codebase, has been audited, and has had security contests over its code. Therefore, the code audit was explicitly commissioned as a delta audit of small scope, over the changes and the differences in the deployment setting.

SETTING & CAVEATS

This audit report mainly covers the contracts of the at-the-time private repository **https://github.com/rhomarkets/Rho_Contract_Audit** of the Protocol at commit `19e4461342b77fa0cc4fbc84d3f8a84cb9aa9856`. The reference commit (trusted base) of the Moonwell project repository is `0abbf3bfce55142c80607fdb7aedf8da52b25954`.

The resolution of the issues was examined at commit

4a5b10fb81e39eef4e7916290ccfacd0b03d67b9 of the at-the-time private repository https://github.com/rhomarkets/Rho_Contracts (branch feat/dedaub-audit)

2 auditors worked on the codebase for 5 days on the following contracts:

```
src/
├── CarefulMath.sol
├── Comptroller.sol
├── ComptrollerInterface.sol
├── ComptrollerStorage.sol
├── EIP20Interface.sol
├── EIP20NonStandardInterface.sol
├── ErrorReporter.sol
├── Exponential.sol
├── ExponentialNoError.sol
├── RErc20.sol
├── RErc20Delegate.sol
├── RErc20Delegator.sol
├── RErc20DelegatorFactory.sol
├── RToken.sol
├── RTokenInterfaces.sol
├── RWethDelegate.sol
├── Rate.sol
├── Unitroller.sol
├── interface/
│   ├── IAddresses.sol
│   └── IWETH.sol
├── irm/
│   ├── InterestRateModel.sol
│   ├── JumpRateModel.sol
│   └── WhitePaperInterestRateModel.sol
├── oracles/
│   ├── AggregatorV3Interface.sol
│   ├── ChainlinkOracle.sol
│   └── PriceOracle.sol
├── rewards/
│   ├── MultiRewardDistributor.sol
│   └── MultiRewardDistributorCommon.sol
│
```

```
├── router/
│   ├── IWETH.sol
│   └── WETHRouter.sol
├── staking/
│   ├── DistributionManager.sol
│   ├── EcosystemReserve.sol
│   ├── EcosystemReserveController.sol
│   ├── StakedRate.sol
│   ├── StakedToken.sol
│   ├── interfaces/
│   │   ├── IDistributionManager.sol
│   │   ├── IERC20Detailed.sol
│   │   ├── IEcosystemReserve.sol
│   │   ├── IStakedToken.sol
│   │   └── ITransferHook.sol
│   └── libraries/
│   ├── DistributionTypes.sol
│   ├── ERC20.sol
│   └── ERC20WithSnapshot.sol
└── utils/
    ├── Addresses.sol
    ├── ChainlinkPriceFeed.sol
    ├── Configs.sol
    ├── Configuration.sol
    ├── ERC20Mintable.sol
    └── WrappedEther.sol
```

The utility files (subdirectory `utils`) are unsafe if deployed (e.g., the ChainlinkPriceFeed allows arbitrary updates) and were treated as test scaffolding for the purposes of the audit. They are, thus, effectively out-of-scope, although inspected.

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed” or “acknowledged” but no action taken, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY:

ID	Description	STATUS
C1	A single empty market with a non-zero collateral factor can be used to drain the protocol	**RESOLVED**

Resolution:

- The protocol has addressed the issue by modifying Compound’s V2 code; the amount that one will receive by calling `redeemUnderlying` will be validated against the current exchange rate. In the context of the attack, the attacker will receive as much underlying as 1 share is worth, even though they will have requested the full amount of donated collateral.
- The protocol will make sure that no empty markets can be used for borrowing assets. This will be achieved by minting a small amount of `rToken` s before setting a non-zero collateral factor.

Moonwell, and subsequently, the Rho protocol, inherit a known Compound V2 rounding issue; an attacker may utilize a `RToken` which has:

- a `totalSupply` of zero and
- a non-zero `collateralFactorMantissa` inside the `Comptroller` contract

in order to borrow all the funds that lie inside the rest of the market contracts.

The attack consists of the attacker first massively inflating the exchange rate of the `RToken` by:

1. Maintaining a balance of 2 `RTokens`

2. Directly transferring a large amount of collateral tokens to the `cToken` contract

The exchange rate inflation naturally allows the attacker to borrow the funds of another market. However, the attacker can then exploit the fact that the rounding direction of the `RErc20::redeemUnderlying` is in favor of the redeemer (attacker):

`RToken::redeemFresh:684`

```
…
} else {
        vars.redeemAmount = redeemAmountIn;

        (vars.mathErr, vars.redeemTokens) =
        divScalarByExpTruncate(redeemAmountIn, Exp({mantissa:
vars.exchangeRateMantissa}));
        …
        }
}
…
```

`CarefulMath:46`

```
function divUInt(uint a, uint b) internal pure returns (MathError, uint) {
        if (b == 0) {
        return (MathError.DIVISION_BY_ZERO, 0);
        }

        return (MathError.NO_ERROR, a / b);
}
```

The function rounds **down** the number of required `rTokens` for the retrieval of the underlying collateral, and this enables the attacker to retrieve all of the donated collateral at the cost of 1 `rToken`(instead of 2). For the protocol, the attacker still holds another `rToken` valued at **the inflated exchange rate**, so the subsequent call to `Comptroller::redeemAllowed` is going to return true.

At that point, the attacker will have emptied one market while retaining all of his donated collateral. Post redemption, the exchange rate of the `rToken` has significantly decreased, making the evaluation of the collateral being far less than the value of the borrowed assets. This allows the attacker to liquidate the remaining `rToken` , which brings the `totalSupply` back to 0 and ultimately allows the attacker to proceed with attacking another market.

To prevent the attack, one can ensure that the totalSupply of an `rToken` is non-zero when the market is updated with a non-zero collateral factor since:

- The greater the `totalSupply`, the **smaller** that the donated amount **must** be in order for the attacker to trigger the rounding issue and redeem all of the donated collateral for 1 less `rToken`.
- The greater the `totalSupply`, the **greater** that the donated amount **must** be so that the exchange rate of the `rToken` is large enough

In order to avoid addressing the issue in Compound V2, we recommend that the protocol be responsible for minting a small amount of `rTokens` before enabling the use of `rTokens` for borrowing.

HIGH SEVERITY:

ID	Description	STATUS
H1	Several search-and-replace errors in ERC20 calculations	**RESOLVED**

Some instances of `add` operations have become `“-”` in the updated code of staking/libraries/ERC20.sol . This is an obvious accounting problem, which should be caught by testing the corresponding part of the functionality.

`ERC20:133`

```
function increaseAllowance(
      address spender,
      uint256 addedValue
) public virtual returns (bool) {
      _approve(
            _msgSender(),
            spender,
            _allowances[_msgSender()][spender] - addedValue
      );
```

ERC20:163

```
function _transfer(
      address sender,
      address recipient,
      uint256 amount
) internal virtual {
  require(sender != address(0), "ERC20: transfer from the zero address");
  require(recipient != address(0), "ERC20: transfer to the zero address");

  _beforeTokenTransfer(sender, recipient, amount);

  _balances[sender] = _balances[sender] - amount;
  _balances[recipient] = _balances[recipient] - amount;
```

ERC20:178

```
function _mint(address account, uint256 amount) internal virtual {
      require(account != address(0), "ERC20: mint to the zero address");

      _beforeTokenTransfer(address(0), account, amount);

      _totalSupply = _totalSupply - amount;
      _balances[account] = _balances[account] - amount;
```

MEDIUM SEVERITY:

M1	Error in the calculation of the staking rewards' distribution index	**RESOLVED (commit fb74280)**

The operator sequence used inside `DistributionManager::_getAssetIndex` seems to diverge from the intended one:

`DistributionManager::_getAssetIndex:279`

```
...
return
      Math.mulDiv(emissionPerSecond, timeDelta, DENOMINATOR) /
      totalBalance +
      currentIndex;

      //@Dedaub: the original sequence of operations is
      //emissionPerSecond * timeDelta * DENOMINATOR /
      //totalBalance +
      //currentIndex;
```

This can effectively break the accounting for the staking rewards, since the amount of accrued rewards inside `stakerRewardsToClaim` will be wrongly scaled:

`DistributionManager::_updateUserAssetInternal:155`

```
...
uint256 newIndex = _updateAssetStateInternal(
      asset,
      assetData,
      totalStaked
);

if (userIndex != newIndex) {
       if (stakedByUser != 0) {
       accruedRewards = _getRewards(stakedByUser, newIndex, userIndex);
       }

       assetData.users[user] = newIndex;
       emit UserIndexUpdated(user, asset, newIndex);
}

return accruedRewards;
```

StakedToken::_updateCurrentUnclaimedRewards:275

```
…
uint256 accruedRewards = _updateUserAssetInternal(
       user,
       address(this),
       userBalance,
       totalSupply()
);
uint256 unclaimedRewards = stakerRewardsToClaim[user] + accruedRewards;
…
```

M2	Protocol operations might be affected by the down-time of third parties	**ACKNOWLEDGED**

Comments:

The protocol attempted to address this issue with the newly introduced oracle contracts, PriceOracle and PriceOracleV2, which were not part of the original audit. Even though a thorough examination of those contracts exceeded the timeframe of the issue resolution period, we have provided the protocol with points regarding:

1. Building an oracle that properly scales the prices to be in the format expected by the Comptroller contract

2. Being aware of the security properties of the source prices (e.g., avoid using AMM spot prices)

We mark this issue as “acknowledged”, although a future iteration of the oracle contracts might resolve this completely.

[Although this issue is potentially important, it is possible that fixing it fully will require non-trivial protocol intervention, and will open the door to other issues–e.g., of transactions conservatively reverting. It is, therefore, reasonable to not address the issue, much as done in Moonwell. However, we bring up the issue to raise awareness, because of its potential for financial loss.]

The protocol intends to use Chainlink’s price oracles in order to retrieve prices for the supported assets. However, the code inside the `ChainlinkOracle` contract does not check the timestamp of the feed’s latest answer:

`DistributionManager:96`

```
function getChainlinkPrice(
       AggregatorV3Interface feed
) internal view returns (uint256) {
       (, int256 answer, , uint256 updatedAt, ) = AggregatorV3Interface(feed)
       .latestRoundData();
       require(answer > 0, "Chainlink price cannot be lower than 0");
       require(updatedAt != 0, "Round is in incompleted state");

       // Chainlink USD-denominated feeds store answers at 8 decimals
       uint256 decimalDelta = 18 - (feed.decimals());
       // Ensure that we don't multiply the result by 0
       if (decimalDelta > 0) {
       return uint256(answer) * (10 ** decimalDelta);
       } else {
       return uint256(answer);
       }
}
```

This means that the protocol might end up consuming stale prices if Chainlink's oracle network ever experiences any downtime.

On a more general note, and since the protocol is expected to be deployed on Scroll, the developers should be aware of the effects that a downtime period in Scroll's sequencer might have. When the sequencer is down, users are still able to enqueue transactions through the L1 bridge contract to be executed when the sequencer is live.

However, during this period price feeds won't be updated since the Chainlink nodes won't be able to call `transmit` on Scroll. Additionally, it seems reasonable to assume that many users won't interact with the L1 bridge contract and they will be unable to interact with the protocol altogether (e.g, they will be unable to provide collateral to their position). These factors might end up leaving the protocol at an inconsistent state and allow knowledgeable parties to enqueue profitable operations against other users and/or the protocol for when the sequencer becomes operational. The protocol might want to consider using Chainlink's sequencer uptime feeds for this.

LOW SEVERITY:

ID	Description	STATUS
L1	Calculation error in view function	**RESOLVED** (staking logic was removed)

A similar issue to H1 can be found in the `StakedToken::getTotalRewardsBalance` function:

`StakedToken::getTotalRewardsBalance:354`

```
      return
            stakerRewardsToClaim[staker] -  // Dedaub: should be "+"
            _getUnclaimedRewards(staker, userStakeInputs);
```

This is an external view function, so it can only throw off external calculations.

L2	Calculation error in currently-unused code	**RESOLVED** (file was removed)

An operator precedence issue appears in WhitePaperInterestRateModel, which is not intended to be used :

WhitePaperInterestRateModel:49

```
function utilizationRate(uint256 cash, uint256 borrows, uint256 reserves)
public pure returns (uint256) {
      // Utilization rate is 0 when there are no borrows
      if (borrows == 0) {
            return 0;
      }

      return (borrows * 1e18 / cash) + borrows - reserves;
        // Dedaub: borrows * 1e18 / (cash + borrows - reserves)
}
```

L3	Non-standard tokens, such as USDT, not supported in sweeping functionality	**ACKNOWLEDGED**

This is an issue inherited from Compound, thus not serious, but one that deployers should know about. The “sweep” functionality in Comptroller does not accept non-standard tokens, such as USDT.

Comptroller::_rescueFunds:974

```
        if (_amount == type(uint).max) {
              token.transfer(admin, token.balanceOf(address(this)));
        } else {
              token.transfer(admin, _amount);
        }
```

The above code expects a high-level return value for function `transfer`, because of the way the function is specified in the IERC20 interface. The code will revert for tokens such as USDT.

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	STATUS
N1	The protocol seems to have a higher degree of centralization than Compound and Moonwell	**ACKNOWLEDGED**

The protocol seems to have removed the governance module present in Moonwell, and it's not apparent whether a governance module will be present at all. In Compound V2, the governance traditionally serves as the admin of the protocol contracts. Among other things, the admin of a Compound V2-like protocol can:

- create/deprecate markets

- set important protocol parameters
- set the risk management unit (Comptroller)
- upgrade the implementation of the contracts

Even if the protocol were to replace the governance module with a multi-sig and not an EOA, the degree of centralization would still naturally increase; In the scenario with no governance, users have to delegate trust to the protocol operator(s), whereas every holder of the governance token would have been able to vote on administrative actions otherwise.

While the admin role has the power to fundamentally control the protocol's logic and funds (as described above), the degree of centralization also increases because of the introduction of functions like `REther::sweepToken` (the function was introduced after the original audit):

REther

```
function sweepToken() external {
       require(msg.sender == admin, ...);

       uint256 balance = address(this).balance;
       require(balance > 0, "No ether left to send");

       (bool sent,) = payable(admin).call{value: balance}("");

       require(sent, "Failed to send Ether");
}
```

The admin can directly call `REther::sweepToken` and receive the entire ETH balance of the contract, a functionality present neither in Compound nor in Moonwell

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Line cloned twice, unused code	**INFO**

A line in the constructor of WhitePaperInterestRateModel has been accidentally copied twice:

WhitePaperInterestRateModel:34

```
constructor(uint256 baseRatePerYear, uint256 multiplierPerYear) {
  baseRatePerTimestamp = baseRatePerYear * 1e18 / timestampsPerYear / 1e18;
  baseRatePerTimestamp = baseRatePerYear * 1e18 / timestampsPerYear / 1e18;
```

ID	Description	STATUS
A2	Inelegant boolean condition	**INFO**

The following boolean condition is inelegant:

Comptroller:1079

```
function isDeprecated(RToken rToken) public view returns (bool) {
      return
            … && borrowGuardianPaused[address(rToken)] == true && …
            // Dedaub: can be just "borrowGuardianPaused[address(rToken)]"
```

ID	Description	STATUS
A3	Considerations regarding the use of `permit` inside `RErc20::mintWithPermit`	**INFO**

As `permit` has no high level return value, calls to `permit` might be turned into silent no-ops if a token that does not implement `permit` has a fallback function that does not revert when a `permit` call is performed. The `WETH` contract on Ethereum is a notorious example of this behavior.

This has no security consequences for `RErc20::mintWithPermit` since the rest of the code will rely on the existing allowance from `msg.sender`:

RErc20:82

```
function mintWithPermit(
        uint mintAmount,
        uint deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
) external override returns (uint) {
        IERC20Permit token = IERC20Permit(underlying);

        token.permit(msg.sender, address(this), mintAmount, deadline, v, r, s);

        (uint err, ) = mintInternal(mintAmount);
        return err;
}
```

RErc20:492

```
function mintInternal(uint256 mintAmount) internal nonReentrant returns (uint256,
uint256) {
        uint256 error = accrueInterest();
        if (error != uint256(Error.NO_ERROR)) {

        return (fail(Error(error), FailureInfo.MINT_ACCRUE_INTEREST_FAILED), 0);
        }

        return mintFresh(msg.sender, mintAmount);
}
```

However, we still wish to bring this to the protocol's attention nonetheless, highlighting a possible subtle point of the integration with specific tokens.

An analogous consideration arises for the tokens that do implement a `permit` function but do not revert when the `permit` operation fails; they instead do not set the specified allowance and neither increase the user's nonce.

Lastly, the protocol should be aware of the fact that anyone may pick up a transaction involving a `RErc20::mintWithPermit` from the mem-pool and front-run the `permit` call on behalf of the owner. This will increase the nonce of the owner, causing the `permit` call inside `RErc20::mintWithPermit` to fail when the transaction of the user is executed. This is not a significant vector of DOS, since the user that got front-run can simply call `RErc20::mint` afterwards. The attacker can only grief the user's gas due to the revert.

A4	Compiler bugs	INFO

The code is compiled with various recent compiler versions. Most files have a floating pragma, `^0.8.23`. Bytecode produced by versions above 0.8.23 is currently not deployable on Scroll: the chain does not have some of the opcodes that the compiler emits. Hence care should be taken to keep the Solidity compiler version to 0.8.23 specifically.. Additionally, we recommend a fixed compiler version for deployment, i.e., no floating pragmas, to remove all ambiguity.

Version `0.8.23`, in particular, has no known bugs at this time.

DISCLAIMER

ABOUT DEDAUB