Smart Contract Audit

coinspect

exactly

coinspect

Exactly Protocol

Smart Contract Audit

V220822

Prepared for Exactly Finance • May 2022

1. Executive Summary

In **April 2022, Exactly Finance** engaged Coinspect to perform a source code review of **Exactly Protocol**. The objective of the four-weeks 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 have been made to the code, the documentation, and the test suite.

Coinspect identified implementation weaknesses that put user funds at risk. Also, some design level threats were observed and suggestions provided in order to mitigate them.

It is recommended to reassess the protocol's security after the issues in this report are fully analyzed and resolved.

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
3	**7**	**1**
Fixed 2	Fixed 6	Fixed 0

Issues EXA-01 to EXA-05 were part of the report for a one-week audit conducted during November 2021.

During August 2022, Coinspect verified most of the issues had been properly addressed by the Exactly team.

2. Assessment and Scope

Exactly is a non-custodial protocol that provides fixed-income solutions for lenders and borrowers by setting interest rates based on supply and demand for credit of each supported token for a certain period.

The audit started on **April 4, 2022** and was conducted on the Git repository at https://github.com/exactly-finance/protocol/releases/tag/0.0.3. The last commit reviewed during this engagement was:

`0685d9ecead2dfad32299d60c73165ef7c02ed87` (**April 13, 2022**).

The scope of the audit was limited to the latest version of the following Solidity source files, shown here with their sha256sum hash:

```
d3a31ec63a2c5e63f0530c4ea354fab1c8dc21286e3a53b58e14d117fb0e95ed  InterestRateModel.sol
711f2ddb876a523a9399effe011ff42c699f0f2c51fd38dc047e4798ded24bf4  FixedLenderETHRouter.sol
9d8e9eb1f21412515233db76dd9f8e94212e1f186749cf4f5065eeee8752d645  FixedLender.sol
bc4ea45d11cafe3c14740cf3b9e9f48c51b94a77d4d44f21c44f3b392783202c  interfaces/IAuditor.sol
c98e2349b1854f3b2caed3c264e4572cb7742463ffe079f1bd62a0d055869275  interfaces/IOracle.sol
4162a6952f8b499f9c8db710742e92cfbce3e31cad09e3c3464327f2e4a045e1  interfaces/IInterestRateModel.sol
a8a3062f02956b6be20cb0bc4614b5a269f94b6b143a0ea36978db1dd2deb4f4  PoolAccounting.sol
80062eba17cae5e066cc5310bacb1ca188f1c1043a754a0d44c016bf642942e5  utils/PoolLib.sol
e2760084b1a58c3fb1d1d0ad23270a236187b6232762da7aa4c5ddc0b36ada2e  utils/TSUtils.sol
e7cb706e5840905b6c410308fc8669462d32774e4274771cfbb5aad91a85f41d  ExactlyOracle.sol
aec77491694301e75d833a20c5c84a4f2a93c4e74fb688dd398b28aa2821928b  Auditor.sol
```

During the assessment the Exactly team fixed several independently discovered issues.

The project is specified to be built with Solidity compiler version 0.8.13. The test suite consists of 678 tests, and all of them pass.

The documentation used as reference during the audit included:

1. Exactly White Paper (April 2022)
2. https://medium.com/@exactly_finance/what-exactly-are-we-building-391d6db2692c

External dependencies:

1. Exactly relies on Chainlink oracles. This is a single point of failure, normally accepted in most DeFi protocolos.
2. Several components inherit from contracts developed by rari-capital (https://github.com/Rari-Capital/solmate)
    a. `ERC4626` and `ERC20`
    b. `safeTransferLib`
    c. `FixedPointMathLib`

The user-facing `FixedLender` contract is implemented as an ERC4626 yield bearing vault token, and it inherits from the Solmate contract located in https://github.com/Rari-Capital/solmate/blob/main/src/mixins/ERC4626.sol.

The administrative roles in the platform can:

1. Create new markets
2. Allow new assets as collateral
3. Change interest rate curve parameters
4. Modify price oracles

These actions are intended to be performed by a multisig wallet acting through a time lock mechanism.

The Exactly team stated that as soon as contracts are deployed, the ownership role `DEFAULT_ADMIN_ROLE` is granted to the `TimelockController` contract (which will have a minimum schedule time of 7 days), leaving the multisig as a proposer and executor of transactions through this `TimelockController`. On the other hand, the role `PAUSER_ROLE` is granted to the multisig, which can call pause and unpause functions at any moment (this feature is only present in `FixedLender`). These deployment and operational security measures were not verified by Coinspect. It is recommended to provide a clear way for users to observe the scheduling of administrative actions in order to allow them to react accordingly.

By design, liquidity providers can only withdraw their assets if they are available (not borrowed) as stated in the Whitepaper. Therefore, **eToken** holders will have the capability of redeeming and receiving their original assets plus its corresponding interests at any time **subject to the available liquidity in the Smart Pool.**

The protocol utilizes an internal precision of 18 decimals. It is important to never add markets with assets with more than that number of decimals as collateral.

The protocol implements a borrow cap that limits the amount of funds that can be borrowed from each `FixedLender`. It is worth noting that by default this cap is set to 0, which leaves this protection mechanism disabled.

Because of gas constraints during liquidity calculations the current design limits the number of markets that can be listed in an `Auditor` contract. Also for the same reason, there is a maximum number of maturities available for each market. Currently it is not possible to remove or de-list a market from the `Auditor`. Markets can be paused if needed, but the `Auditor` could become full of paused markets.

Users' liquidity is calculated in the context of each `Auditor` and its registered markets. There can be as many `Auditor` contracts as desired, but liquidity cannot be calculated across different `Auditors`.

Because a user's liquidity is calculated taking into account all the assets (borrows and Smart Pool deposits) listed in one `Auditor`, the security of all the assets from all `FixedLenders` (markets) in that contract is dependent on one another. If only one of those assets introduces a vulnerability, or if the associated price oracle can be manipulated (for example because of its reliance on a low liquidity pool as source for the price), an attacker would be able to steal from all the other markets controlled by the same `Auditor`. It is important to be careful when incorporating new assets markets and price oracles into the platform.

As with other protocols, the security of the funds will depend on several configuration parameters which have to be tuned by the platform administrative roles to adapt to the changing scenarios. It is imperative that healthy minimum collateralization ratios are kept for each market. It is recommended that each asset that is accepted as collateral is fully evaluated, to determine its associated risk and to make sure its configuration parameters are tuned accordingly. For example, riskier assets should be set with a lower debt ceiling to prevent the system from accumulating too much debt in a dangerous asset.

3. Summary of Findings

Id	Title	Total Risk	Fixed
EXA-06	Protocol can accumulate debt because of unprofitable liquidations	High	✔
EXA-07	Liquidity check bypass allows attackers to drain protocol funds	High	✔
EXA-08	Some users will not be able to redeem funds when protocol has debt	High	⏳
EXA-09	Interest rate model approximation error	Medium	✔
EXA-10	Cascading liquidations speed limiting factors	Medium	⏳
EXA-11	Attackers can manipulate interest rate to obtain cheap borrows	Medium	✔
EXA-12	Rounding to zero allows bypassing allowance and liquidity checks	Medium	✔
EXA-13	Allowance reset could result in user funds lost	Medium	✔
EXA-14	Liquidations revert when the liquidator has shortfall	Medium	✔
EXA-15	Chainlink pair rates are assumed to always have 8 decimals	Medium	✔
EXA-16	Attackers can block liquidity providers withdrawals (griefing attack)	Low	⏳
EXA-17	Listed FixedLenders contracts are allowed to seize all user funds	Info	✔
EXA-18	Protocol does not verify expected amounts are transferred by third party contracts	Info	⏳

EXA-19	Incorrect comment in FixedLender regarding beforeRepayMP	Info	✔
EXA-20	Unnecessary addition in PoolAccounting	Info	✔
EXA-21	Missing pool state validation in liquidate function	Info	⏳
EXA-22	Missing sanity checks in some critical protocol parameters setters	Info	⏳

4. Detailed Findings

EXA-06 Protocol can accumulate debt because of unprofitable liquidations

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

Description

Attackers can create many small positions in order to create debt in the system. Exactly protocol lacks a minimum position size. As a consequence, some of these positions incentive for liquidators will not be enough when compared to the gas expense for executing the liquidation.

This is further aggravated because:

1. The close factor also affects these calculations, as the liquidation can only seize a part of the collateral in the position (e.g., 50%).
2. Liquidators can only liquidate debt from one `FixedLender` at a time. A user's debt can be fragmented between all the markets, requiring many liquidations until the position is back to being properly collateralized.

Coinspect auditors observed the shortfall and liquidity calculation is implemented using a loop that traverses all markets and all maturities.

Upon consultation, the Exactly team prepared a test to calculate the gas cost of a liquidation in an adverse scenario. The test estimates 1,963,773 gas (with gas price = 30 gwei ~= 185 USD with ETH = 3100 USD) will be required to liquidate a user position with debt in 4 markets with 216 maturities. This would get worse as more markets are added to the platform. **However, the team clarified that even if it is supported by the protocol, it is not planned to ever allow that many maturities.**

Recommendation

Enforce a minimum position size to guarantee liquidators are always properly incentivized. Calculate the cost of liquidating positions with different gas prices and close factors.

Status

This issue was fixed by modifying the liquidation process so this attack is more costly and less likely. No minimum position size was implemented.

EXA-07 Liquidity check bypass allows attackers to drain protocol funds

Total Risk	Impact	Location
High	High	`FixedLender.sol`
Fixed ✓	Likelihood High	

Description

Attackers can drain funds from the protocol and leave it with uncollateralized debt.

Coinspect observed the current `beforeWithdraw` function implementation is flawed and can be bypassed when the `msg.sender` and the funds owner are different. **This enables attackers to drain funds from the protocol and leave it with uncollateralized debt.**

As explained in the previous issue, the `beforeWithdraw` hook is responsible for checking if an account has no shortfall (more debt than collateral). This virtual function in the `ERC4626` contract imported from the rari-capital project is intended to allow the derived contracts to perform the checks necessary before funds are withdrawn. Specifically, this hook is called from the `withdraw` and `redeem` functions.

The `beforeWithdraw` function gets passed 2 parameters

1. Number of assets
2. Number of shares

The account from where the funds are being withdrawn is not passed down. As a result, as the previous code snippet in the previous issue shows, the `msg.sender` is the account whose liquidity is verified.

However, both `withdraw` and `redeem` functions allow the caller to retrieve funds from a different account if this account has given it the corresponding allowance:

```
   function redeem(
       uint256 shares,
       address receiver,
       address owner
   ) public virtual returns (uint256 assets) {
```

As a consequence, attackers can approve another account in their control, and use this account to redeem/withdraw their funds, bypassing the liquidity checks in place.

Recommendation

Verify the liquidity of the owner of the funds being withdrawn.

Status

This issue was fixed by adding account shortfall validation on the `withdraw` and `redeem` functions of the `FixedLender.sol` contract.

EXA-08 Some users will not be able to redeem funds when protocol has debt

Total Risk	Impact	Location
High	High	`FixedLender.sol`
Fixed ⏳	Likelihood High	

Description

When liquidations are not able to recover funds fast enough, the protocol is left with outstanding debt that is not redistributed between depositors. As a result, those last users that attempt to withdraw their deposits won't be able to do it, as the protocol liquidity will not suffice.

After a price decline in one asset, some positions will be liquidated. At some point, it could happen that liquidators seize all collateral in those positions, but the debt has not been fully paid for. In that case, the protocol as a whole has outstanding debt, and liquidators have no incentive to repay the debt.

The current implementation does not consider this protocol debt: the assets per market share is not updated to reflect this fact. As a consequence, when users withdraw their funds, they are paid in full. However, the last users that attempt to withdraw will find no liquidity is left.

Recommendation

Consider equally redistributing the outstanding debt between the market shares.

Status

In progress.

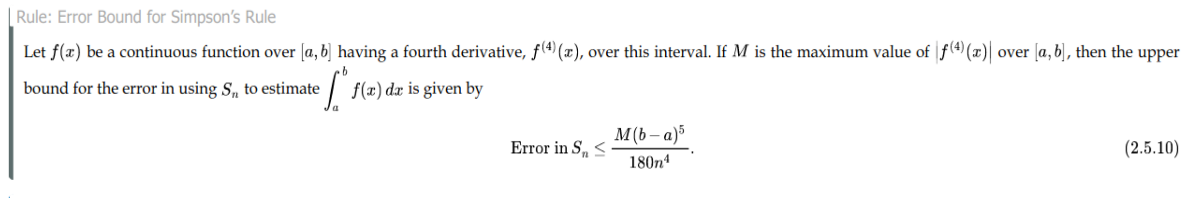
EXA-09 Interest rate model approximation error

Total Risk	Impact	Location
Medium	High	`InterestRateModel.sol`
Fixed ✔	Likelihood Low	

Description

`InterestRateModel` utilizes Simpson's rule to numerically integrate Exactly's borrow rate function. Error bounds of this method could be very high on a rational function:

From: Numerical Integration - Midpoint, Trapezoid, Simpson's rule - Mathematics LibreTexts



Rule: Error Bound for Simpson's Rule

Let $f(x)$ be a continuous function over $[a,b]$ having a fourth derivative, $f^{(4)}(x)$, over this interval. If M is the maximum value of $|f^{(4)}(x)|$ over $[a,b]$, then the upper bound for the error in using S_n to estimate $\int_a^b f(x)\,dx$ is given by

$$\text{Error in } S_n \leq \frac{M(b-a)^5}{180n^4}. \qquad (2.5.10)$$

In our case n=4 so the denominator is a small number.

Exactly's simplified rate curve is defined as:

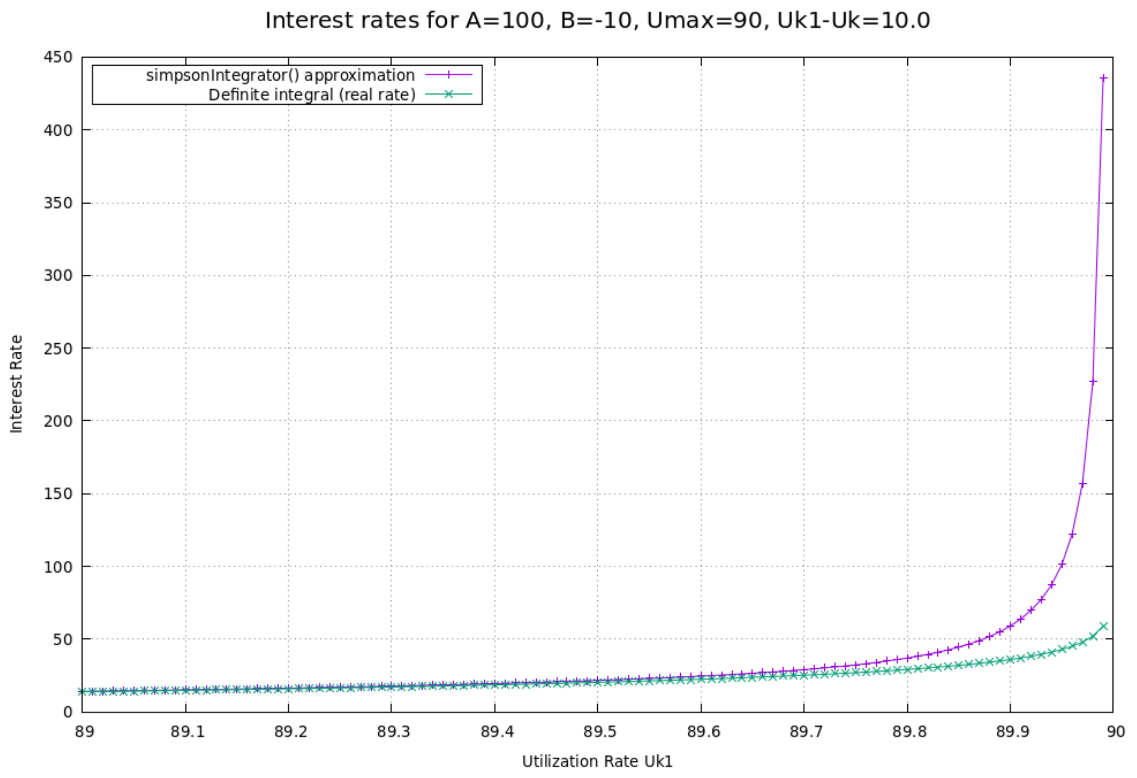
$$f(x) = \frac{A}{ut} + B$$

Then the fourth derivative is:

$$f^{(4)}(x) = \frac{25A}{ut^5}$$

As *ut* approaches zero, maximum value M approaches infinity, so Error in Sn ≤ infinity, meaning the Simpson's Rule error close to the asymptote is unbounded (too high).

As a result, the estimation of the interest rate will increase asymptotically before the actual interest rate function. This will ultimately cause much higher borrow interest rates than specified in the whitepaper, and excessive penalty fees at early withdrawal.



The trapezoid integrator considers the rightmost-point for calculations, and in Exactly's equation:

```
rate = curveParameterA / (uMax-utilizationRate) + curveParameterB
```

When `the denominator` is zero (utilizationRate approaches umax, the maximum utilization rate set), the rate is infinite. The trapezoidal integrator will return a value much higher than the real integral for this function.

The figure above depicts an example where the estimated interest rate rapidly diverges from the real interest rate. In the example, at the utilization interval 79.9-89.9, the simpsonIntegrator() function returns an interest rate of 59, while the real interest rate calculated with the definite integral returns 37, and this difference only increases when approaching the max utilization rate of 90%.

Recommendation

Perform the calculation using a definite integral instead of a numerical approximation. This solution requires the implementation of a fixed-point logarithm.

Status

This issue was fixed by replacing the integral approximation algorithm with an exact calculation.

EXA-10 Cascading liquidations speed limiting factors

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

Description

A few factors, described below, limit the protocol's liquidation capacity. The inability to liquidate assets in a timely fashion would result in growing outstanding debt and deposits in other assets locked in the platform until the situation is repaired.

This would further be aggravated if the mempool is full, for example, as other protocols and users compete to refund their positions during a market crash.

Based on the scenario described in the previous issue: with a current block gas limit of approximately 30058562 gas and a worst-case scenario of 1963773 gas cost for each liquidation, a maximum of 15 liquidations per block can take place (when the mempool is empty).

The Exactly team clarified that the number of maturities planned to be used will be much lower (around 24) when the protocol is deployed.

The issue here described is common to all protocols in the Ethereum network and can not be fully mitigated. Exactly protocol's liquidation speed is currently limited by:

1. The close factor as the liquidation can only seize a part of the collateral in the position (e.g., 50%).
2. Liquidators can only liquidate debt from one `FixedLender` at a time. A user's debt can be fragmented between all the markets, requiring many liquidations until the position is back to being properly collateralized.

A few suggestions are provided below in order to help mitigate this risk.

Recommendation

Coinspect recommends evaluating the following potential ways of improving the speed of liquidations in order to further bulletproof the protocol:

1. Decrease the gas cost of the calculation of a user’s liquidity (are there any values that can be cached?)
2. Limit the number of markets a user can participate in.
3. Consider adding some form of batched liquidations (e.g., allow liquidating a user's debt from multiple fixed lenders in one transaction).

Status

In progress.

EXA-11 Attackers can manipulate interest rate to obtain cheap borrows

Total Risk	Impact	Location
Medium	High	`PoolAccounting.sol`
Fixed ✓	Likelihood Medium	

Description

Attackers can deposit to a market's `SmartPool` to decrease the pool utilization rate in order to decrease the interest rate and then borrow at a cheaper rate.

After that, the attackers can immediately withdraw from the `SmartPool` if it has enough assets. The assets in the `SmartPool` must be enough to fulfill the borrow and also to allow the attacker to recover his first deposit. Alternatively, this could be done using the MP to inflate the utilization rate but would require leaving the funds until next maturity.

The `borrowMP` function in `PoolAccounting` calculates interest rate (fee) based on the utilization rate, and passes the `smartPoolTotalSupply` to the`InterestRateModel` `getRateToBorrow` function:

```
borrowVars.fee = amount.fmul(
  interestRateModel.getRateToBorrow(
    maturity,
    block.timestamp,
    amount,
    pool.borrowed,
    pool.supplied,
    smartPoolTotalSupply
  ),
  1e18
);
```

For example, attackers could:

1. Flash loan (not really required, but helpful depending on the pool size)
2. Call `deposit()`
3. Call `borrowAtMaturity()`
4. Call `withdraw()`
5. Payback flash loan

As a result, the attacker would get a cheap loan. Coinspect did not fully evaluate the profitability of this attack in different scenarios.

Related concepts in the Whitepaper:

1. "*Each new deposit generates an increase in the liquidity for the specific Maturity Pool, reducing its utilization rate and its fixed interest rate for a new loan.*"
2. "*A key aspect of utilizing the amount of money 'post-transaction' to calculate the interest rate is to avoid users taking all the liquidity (driving Utilization Rate to 100%) while having locked in a very low rate at a certain point in time.*"

Recommendation

Re-evaluate how the utilization rate affects the interest rate calculations.

Status

This issue was fixed by using the average utilization rate instead of the current utilization rate.

EXA-12 Rounding to zero allows bypassing allowance and liquidity checks

Total Risk	Impact	Location
Medium	High	`FixedLender.sol`
Fixed ✓	Likelihood Low	

Description

Loss of precision under some circumstances will cause the conversion of a non-zero number of assets to zero shares (and vice versa). Attackers can utilize this to bypass critical checks and steal funds.

The ERC4626 `ConvertToAssets` and `ConvertToShares` functions use the `FixedPointMathLib` `mulDivDown` function which rounds-down the result as per EIP-4626.

The following functions in the `FixerLender` contract fail to consider this possibility: `transfer`, `transferFrom`, `borrowAtMaturity`, `withdrawAtMaturity`.

As a consequence, critical checks are bypassed when the 0 value is used.

For example, in the case of the `withdrawAtMaturity` function in the `FixedLender` contract, this can be used to bypass the allowance check. **An attacker can withdraw assets and/or borrow funds using collateral owned by another account.**

For demonstration purposes, we consider a FixedLender with an asset having 6 decimals:

1. We call `withdrawAtMaturity(0.000001)` to withdraw 0.000001 assets.
2. This results in a call for `convertToShares(0.000001)`.
3. The operation is `shares=(0.000001*totalSupply)/totalAssets()`

If `totalAssets() > totalSupply`, then the operation `convertToShares(0.000001)` will round down and return zero.

Then this value is used to validate the funds' owner allowance:

```
   if (msg.sender != owner) {
     uint256 allowed = allowance[owner][msg.sender]; // saves gas for limited
approvals.

     if (allowed != type(uint256).max) allowance[owner][msg.sender] =
                         allowed - convertToShares(assetsDiscounted);
   }
```

As a result, the owner's share allowance will never be decreased, and the operation will not revert if the allowance is zero, allowing the msg.sender to always collect 0.000001 assets from any owner The example 0.000001 is the minimum possible value. This value increases proportionally with the ratio between `totalSupply` and `totalAssets()`. For example, if this ratio is 200:1, each transaction can collect 0.0002 assets.

This attack is only practical for low decimals and high-price assets. Otherwise, the cost of the transaction will likely be higher than the amount transferred.

The ERC4626 contract imported from the rari-capital project prevents the rounding from being exploited by checking for 0 each time the functions that round down are used:

```
  function deposit(uint256 assets, address receiver) public virtual
                                          returns (uint256 shares) {
      // Check for rounding error since we round down in previewDeposit.
      require((shares = previewDeposit(assets)) != 0, "ZERO_SHARES");
```

Recommendation

In functions that perform round down, check the return value is not zero.

Status

This issue was fixed by using the functions `previewMint`/`previewWithdraw` that automatically rounds up results, preventing rounding errors.

EXA-13 Allowance reset could result in user funds lost

Total Risk	Impact	Location
Medium	High	`FixedLender.sol`
Fixed ✓	Likelihood Low	

Description

User token allowance is reset during liquidations. This could result in lost funds in certain use case scenarios.

When a liquidation occurs, the `_seize` function is called. This function is implemented in 3 steps:

1. Caller validations
2. “Simulated” allowance reset from liquidated borrower to `msg.sender`
3. ERC4626’s `redeem` is called

```
uint256 shares = previewWithdraw(assets);
allowance[borrower][msg.sender] = shares;

// That seize amount diminishes liquidity in the pool
redeem(shares, liquidator, borrower);

emit AssetSeized(liquidator, borrower, assets);
```

The allowance is set in order to simulate a redeem from the liquidated account with the liquidator as the funds receiver.

However, if an existing allowance exists, it will be reset. This is not expected by the liquidated account and could result in unknown side effects.

For example:

1. Address1 gives allowance to BotAddress to operate with all of its assets. BotAddress2 could be a smart contract/service designed to invest other accounts funds. Address1 is usually off-line or a cold wallet.
2. BotAddress also is capable of triggering liquidations in the protocols it supports in order to gain additional profits for its users.
3. BotAddress observes Address1 is in debt and liquidates part of its position. As a consequence, `allowance[Address1][BotAddress]` is reset to 0.
4. BotAddress observes Address1 shortfall and deposits more collateral to avoid further liquidations. However, this transaction fails.

Recommendation

Do not reset the user allowance. The seized amount can be added to the current allowance so after the seize executes, the allowance is the expected one.

Status

This issue was fixed by modifying the `_seize` function so it no longer uses allowance to transfer shares.

EXA-14 Liquidations revert when the liquidator has shortfall

Total Risk	Impact	Location
Medium	Medium	`FixedLender.sol`
Fixed ✓	Likelihood Medium	

Description

The `redeem` function is reused as part of the liquidation flow. As a consequence, the `_seize` function reverts the liquidation if the liquidator itself is underwater. This is not the intended behavior and could prevent liquidations or result in funds being lost in certain case scenarios as described below.

The `_seize` function is called as part of the liquidation process. As shown in the previous issue, this function is implemented in 3 steps:

1. Caller validations
2. "Simulated" allowance reset from liquidated borrower to `msg.sender`
3. ERC4626's `redeem` is called

This internal function can be reached from 2 different execution flows:

1. Called directly by the same contract: this takes place when the collateral and borrow `FixedLenders` are the same. The `msg.sender` continues to be the liquidator's address that started the liquidation when `_seize` is reached.

2. Called via the external `seize` function: this occurs when the markets are different and an external call is performed to the external `seize` function in the collateral market FixedLender. In this case, the msg.sender is passed from the external `seize` to the internal `_seize` as its `seizerFixedLender` parameter. The `msg.sender` is the original `FixedLender` where the liquidation was initiated.

It is important to note the `msg.sender` is different depending on how the `_seize` function is reached.

Coinspect noticed that when the `redeem` function is called, the `beforeWithdraw` hook is invoked as in a regular redeem.

```
   function redeem(
       uint256 shares,
       address receiver,
       address owner
   ) public virtual returns (uint256 assets) {
       if (msg.sender != owner) {
           uint256 allowed = allowance[owner][msg.sender]; // Saves gas for limited
approvals.

           if (allowed != type(uint256).max) allowance[owner][msg.sender] = allowed -
shares;
       }

       // Check for rounding error since we round down in previewRedeem.
       require((assets = previewRedeem(shares)) != 0, "ZERO_ASSETS");

       console.log("redeem() calling before withdraw");
       beforeWithdraw(assets, shares);

       _burn(owner, shares);

       emit Withdraw(msg.sender, receiver, owner, assets, shares);

       asset.safeTransfer(receiver, assets);
   }
```

The `beforeWithdraw` is a virtual method in the ERC4626 contract, that is implemented by Exactly's `FixedLender`. This hook is intended to guarantee that the protocol is not left with debt when assets are being withdrawn. However, the check is performed on the `msg.sender`:

```
function beforeWithdraw(uint256 assets, uint256) internal override {
  //console.log("beforeWithdraw() validating shortfall for ", msg.sender);

  auditor.validateAccountShortfall(this, msg.sender, assets);
```

As mentioned before, the `msg.sender` is different depending on how this check was reached:

1. When `msg.sender` is the liquidation initiator `FixedLender`, this market liquidity will be evaluated. *This check is unnecessary, as the market's should not have debt.*
2. When `msg.sender` is the address that called `liquidate` in the FixedLender, its liquidity will be calculated and evaluated.

As a consequence, liquidations started from liquidators with a shortfall are not allowed and reverted. This prevents a liquidator with debt from obtaining funds to repay it and, for example, it could affect other contracts with investing strategies that rely on the Exactly framework.

Recommendation

Coinspect recommends re-evaluating the current `liquidate->_seize->allowance->redeem` execution flow in order to improve it and remediate this and the other issues. In particular, consider not relying on the `redeem` function for the seize functionality.

Status

This issue was fixed by modifying the `_seize` function so it no longer uses `redeem` but it directly calls `safeTranfer`.

EXA-15 Chainlink pair rates are assumed to always have 8 decimals

Total Risk **Medium**	Impact High	Location `ExactlyOracle.sol`
Fixed ✔	Likelihood Low	

Description

If a Chainlink pair with ETH as base price is used, the precision scaling will be incorrect and this would result in lost funds.

The `ExactlyOracle` contract gets passed a `baseCurrency` in the constructor.

Chainlink pair rates are assumed to have 8 decimals as defined by the `ORACLE_DECIMALS` constant. This value is used by the scaling function `_scaleOraclePriceByDigits`.

However, Chainlink pairs with ETH as base price have 18 decimals.

Recommendation

If the oracle contract is never intended to be used with ETH as `baseCurrency` it should be documented or forbidden in the constructor, as a mistake here could result in severe consequences.

Alternatively, use `decimals()` provided by Chainlink instead of using a hardcoded value.

Status

This issue was fixed by documenting that the base currency used in the Chainlink oracle is USD.

EXA-16 Attackers can block liquidity providers withdrawals (griefing attack)

Total Risk	Impact	Location
Low	Medium	`FixedLender.sol`
Fixed ⏳	Likelihood Low	

Description

Coinspect identified an attack that prevents `SmartPool` depositors from withdrawing their funds. This attack is only possible if the perpetrators are willing to spend some funds in order to harm the Exactly protocol and its users.

Even though liquidity providers know their ability to withdraw their funds depends on the pool liquidity (e.g., the funds have not been borrowed), during the attack the funds might be available but they would still not be able to retrieve them.

This issue is not directly profitable for the attacker, and because of that it is only considered as a low risk issue. However, attackers might be able to profit from this issue, for example in the secondary market where market shares are traded.

The attack goal is to block SP withdrawals with a `borrow()`. The attacker can borrow (at next maturity) if she observes a `withdraw()` TX in the mempool in order to prevent depositors from removing funds from SP. For this to be possible, the attacker needs enough collateral in the `Auditor` to succeed (those can be in any other asset listed in the same `Auditor`).

One possible attack would be:

1. Attackers setup by providing assets to Market X
2. User1 provides assets to Market A SP
3. User1 wants to withdraw and posts a `withdraw()` TX
4. Attackers sandwich (possibly using a MEV bot) User1's TX with
    a. Pre: borrow in Market A (closest maturity)

        i. Reserve factor will be left in the SP (see note below on how the attack could be improved in this respect)
        ii. Their assets in Market X allow the borrow
    b. **User1's withdraw() reverts because lack of liquidity**
    c. Post: repay in Market A (closest maturity)

As a consequence:

1. Users can't remove their liquidity from the market and the funds are locked forever.
2. LPs are discouraged from depositing in the SP.
3. Then users can't borrow.
4. Also, this could impact secondary market prices of Exactly' eTokens.

Additional notes:

1. `deposit/withdraw()` to SP does not imply any cost (besides gas)
2. The `smartPoolReserveFactor` prevents borrowing in `PoolAccounting` `borrowMP()` which leaves at least 20% (max) in the SP unborrowed. However, the attacker could withdraw that 20% (if he previously made the deposit) and leave no reserve. This would prevent future borrows until more assets are deposited into SP.

The following mitigating factors apply:

1. Advanced users could hide their `withdraw()` TX (at an extra cost).
2. SP are not guaranteed the availability of their funds, so this can be considered a known risk.
3. The reserve factor acts as a partial mitigation, but we believe it could be bypassed as described above.

The attack is more likely to succeed in pools with low liquidity. Coinspect did not perform a full cost/benefit analysis of this issue.

Recommendation

Allow users to withdraw as much as possible if not enough assets are available instead of reverting the transaction.

Status

Exactly does not consider this an issue because it is not profitable for the attacker.

EXA-17 Listed FixedLenders contracts are allowed to seize all user funds

Total Risk	Impact	Location
Info	-	`Auditor.sol` `FixedLender.sol`
Fixed ✔	Likelihood -	

Description

The `FixedLender` external `seize` function and the `Auditor` `seizeAllowed` function are used during the liquidation of a position. They allow any listed market to seize all funds from other `FixedLenders` without further checks.

The function `seizeAllowed` performs the following validations:

1. Borrower is different than liquidator
2. Both markets are listed in the `Auditor`

Moreover, the decoupling between `liquidate` validations and the external `seize function` (required by the reentrancy protection mechanism) is dangerous. ***Even though the risk of a rogue administrative action is mitigated by the timelock, a new exploitable `FixedLender` contract could be introduced (unknowingly or purposely) in the platform in the future (e.g., to support new features).***

Also, if this happens, it is currently not possible to delist a market from the `Auditor` in order to block a market from continuing to seize funds. However, the `FixedLender` could be paused, which currently prevents calls to the `seize` function.

Recommendation

Consider an alternative design and/or add more checks to completely rule out all abusive `seize` scenarios.

Status

This is not currently exploitable.

EXA-18	Protocol does not verify expected amounts are transferred by third party contracts	
Total Risk **Info**	Impact -	Location `FixedLender.sol` `ERC4626.sol`
Fixed ⏳	Likelihood -	

Description

Exactly smart contracts assume external contracts (e.g., assets deposited as collateral in the markets) always transfer the expected amount.

User-exposed functions do not verify if the asset's `safeTransferFrom` resulted in the expected amount being transferred. Instead, if the call does not revert, they assume the total amount was transferred.

The `FixedLender` contract could be abused to mint more tokens than the amount corresponding to the collateral actually deposited. In a similar fashion other functions handling critical fund transfers could be exploited as well. The total amount is assumed to be transferred and it is used to calculate the number of shares to mint to the depositor.

There are tokens that transfer less than the specified amount. For example, the USDT token's `transfer` and `transferFrom` functions (Tether: USDT Stablecoin | 0xdac17f958d2ee523a2206206994597c13d831ec7) deduct a fee for each transfer if a fee percentage is configured. While it is not configured to take fees right now, this could change in the future.

Not trusting the external contract is a defense in depth mechanism. If it is decided to trust the external contracts, then it is critical to establish a process to whitelist new assets and markets, and to continuously follow up their status: document asset behavior, be careful with contract updates and/or modifications of their configuration parameters (e.g., fees), as these modifications could impact Exactly's safety.

Recommendation

It is advised to check the balance of the contract before and after the `transferFrom` call is performed to determine the exact amount that was received to bulletproof Exactly's pools for future updates.

Clearly document the possibility of tokens transferring less than expected. Make sure this is considered when new assets are added to the platform, and follow up all assets updates and configuration changes.

Status

Exactly decided to not address this issue since it was evaluated as not currently exploitable.

EXA-19 Incorrect comment in FixedLender regarding beforeRepayMP

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

Description

In the `_repay` function, the comment below is not accurate, as the `beforeRepayMP` function does not exist.

```
/// @dev Internal repay function, allows partial repayment.
/// Should be called after `beforeRepayMP` or `liquidateAllowed` on the auditor.
```

Recommendation

Remove the comment.

Status

This issue was fixed by removing the incorrect comment.

EXA-20 Unnecessary addition in PoolAccounting

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

Description

The following addition can be replaced with an assignment:

```
earningsSP += pool.accrueEarnings(maturity, block.timestamp);
```

This issue has no impact besides the gas expense.

Recommendation

Replace the addition with an assignment.

Status

This issue was fixed by following the recommendation, and replacing the addition with an assignment.

EXA-21 Missing pool state validation in liquidate function

Total Risk	Impact	Location
Info	-	`FixedLender.sol`
Fixed ⏳	Likelihood -	

Description

The `validateRequiredPoolState` function is not called in the `liquidate` function. As a consequence, an invalid maturity date is passed down to internal functions.

The rest of the user exposed functions that receive a maturity as a parameter validate it:

```
    // reverts on failure
    TSUtils.validateRequiredPoolState(maxFuturePools, maturity, TSUtils.State.VALID,
TSUtils.State.NONE);
```

However, the `liquidate` function is missing this check. Coinspect recommends reverting early when the parameter provided is incorrect.

Recommendation

Add the missing check to the `liquidate` function.

Status

This issue won't be fixed as Exactly considers that not validating the pool state has no effect.

EXA-22	Missing sanity checks in some critical protocol parameters setters	

Total Risk
Info
	Impact
	-
		Location
		`Auditor.sol`
		`InterestRateModel.sol`
		`FixedLender.sol`

Fixed ⏳	Likelihood -	

Description

Coinspect observed that some critical parameters that affect the security of the protocol funds are not always validated.

For example, the `setCollateralFactor` function in the `Auditor` contract checks the factor's range to be correct. However, the `Auditor` `enableMarket` function does not, allowing the addition of a market with an invalid collateral factor that could put user funds at risk.

The same situation is repeated for the `liquidationIncentive` parameter.

In the `InterestRateModel` contract the `spFeeRate` parameter range is validated only in the setter, but not in the constructor. Additionally, none of the curve parameters are range-validated: e.g., a `maxUtilizationRate` >100% can be set

Again, this is also true for the `accumulatedEarningsSmoothFactor` and `maxFuturePools` parameters of the `FixedLender` contract.

Recommendation

Coinspect recommends making sure validations are consistent to prevent incorrect values that could result in lost funds.

Status

The Exactly team clarified this was done on purpose. As the protocol has many parameters/factors that can be set from outside (`ADMIN` role), Exactly avoided placing the validations when parameters are first initialized in constructors or in the `enableMarket` function that is only called once when adding a new asset to the protocol, as they believe that they shouldn't get mistaken when deploying nor when enabling a new market. Exactly acknowledged that the case that an invalid collateral factor in a new market can still harm users' funds in other markets should be considered, though.

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.