SHERLOCK

Security Review For Extra Finance

Private Best Efforts Contest Prepared For: **Extra Finance**
Lead Security Expert: **A2-security**
Date Audited: **November 20 - December 1, 2024**

Introduction

Extra Finance is a leveraged yield farming (LYF) protocol built within the Superchain ecosystem. the contest is help further improve the robustness of the protocol.

Scope

Repository: ExtraFi/extra-contracts

Branch: main

Audited Commit: db1e3e20a45ae98049431475340f3015b36e659f

Final Commit: 4302ab47017f5b39de7a6aedd56931607aca7b8b

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
1	2

Issues Not Fixed or Acknowledged

High	Medium
0	0

Security experts who found valid issues

jennifer37
A2-security
KupiaSec
novaman33
0xc0ffEE
araj
smbv-1923
zarkk01
ck
vinica_boy
TessKimy

Issue H-1: In setReward(), block.timestamp is used instead of startTime at `lastUpdateTime`

Source: https://github.com/sherlock-audit/2024-11-extra-finance-v1-judging/issues/2

Found by

0xc0ffEE, A2-security, KupiaSec, araj, jennifer37, novaman33, smbv-1923

Summary

In setReward(), block.timestamp is used instead of startTime at `lastUpdateTime`. This results in reward distribution for the time, reward was not allocated for.

Root Cause

In setReward(), owner can set rewards for present as well as future. Issue arises when owner sets rewards for future because at `rewardData[rewardToken].lastUpdateTime`, block.timestamp is used instead of startTime.

Internal pre-conditions

None

External pre-conditions

None

Attack Path

For easier calculation, duration is taken in hours

1. Suppose its 1PM, Alice staked 5e18 tokens.
2. And owner wanted to distribute a rewardToken(amount = 100e18) starting at 2PM till 6PM ie duration = 4 hours. Therefore, rewardRate = 100e18 / 4 hours = 25e18 tokens/hour & lastUpdateTime = 1PM
3. Now, there should not be any reward from 1PM to 2PM because reward started from 2PM.

4. However, Alice will receive reward for 1PM to 2PM also

Impact

Users will receive reward for the time, reward was not allocated for. In above case, for 1PM to 2PM

PoC

```
function test_rewardWillDistributeForWrongTime() public {
       //Alice deposited 5e18 tokens at 1PM
       address alice = makeAddr("Alice");
       vm.startPrank(alice);
       uint256 amount = 5e18;
       stakedToken.mint(alice, amount);
       stakedToken.approve(address(stakingContract), amount);
       stakingContract.stake(amount, alice);
       vm.stopPrank();

       //Owner set reward at 1PM but starting from 2PM
       vm.startPrank(owner);
       MockedMintableERC20 rewardToken = new MockedMintableERC20("Rewards",
↪  "Rewards");

       uint256 startTime = block.timestamp + 1 hours;
       uint256 endTime = startTime + 4 hours;
       uint256 totalRewards = 100e18;

       rewardToken.mint(owner, totalRewards);
       rewardToken.approve(address(stakingContract), totalRewards);
       stakingContract.setReward(address(rewardToken), startTime, endTime,
↪  totalRewards);
       vm.stopPrank();

       //Alice is claiming after 1 hour ie at 2PM
       vm.warp(block.timestamp + 1 hours + 1);
       vm.roll(block.number + 1);

       vm.startPrank(alice);
       stakingContract.claim();
       //balanceOf alice should be 0 but it is ~25e18 & stakingContract is ~ 75e18
       console.log("balanceOf alice: ", rewardToken.balanceOf(alice));
       console.log("balanceOf stakingContract: ",
↪  rewardToken.balanceOf(address(stakingContract)));
       vm.stopPrank();
   }
```

Result:

```
Ran 1 test for tests/lending-pool/StakingRewards.t.sol:StakingRewardsTest
[PASS] test_rewardWillDistributeForWrongTime() (gas: 992944)
Logs:
  balanceOf alice:  25006944444444442840
  balanceOf stakingContract:  74993055555555557160
```

Mitigation

Use startTime instead of block.timestamp for lastUpdateTime

```
    function setReward(address rewardToken, uint256 startTime, uint256 endTime,
↪   uint256 totalRewards)
        public
        onlyOwner
        nonReentrant
        updateReward(address(0))
    {
...
        startTime = Math.max(block.timestamp, startTime);

        rewardData[rewardToken].startTime = startTime;
        rewardData[rewardToken].endTime = endTime;
-       rewardData[rewardToken].lastUpdateTime = block.timestamp;
+       rewardData[rewardToken].lastUpdateTime = startTime;
...
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/ExtraFi/extra-contracts/pull/2

Issue M-1: `LendingPool::setBorrowingRateConfig` does not update the `currentBorrowingRate` because it doesn't call `ReserveLogic::updateState` and `ReserveLogic::updateInterestRates`.

Source: https://github.com/sherlock-audit/2024-11-extra-finance-v1-judging/issues/35

Found by

A2-security, KupiaSec, jennifer37, vinica_boy, zarkk01

Summary

The missing calls to `updateState` and `updateInterestRates` from `LendingPool::setBorrowingRateConfig` will result to the update to not having an immediate effect on the `currentBorrowingRate` and instead waiting for the next action to update it.

Root Cause

In `LendingPool::setBorrowingRateConfig`, the code is trying to update the current configuration for the `borrowingRates` depending on the `utilisationRate` of the reserve. Let's see the implementation of the function :

```
function setBorrowingRateConfig(
    DataTypes.ReserveData storage reserve,
    uint16 utilizationA,
    uint16 borrowingRateA,
    uint16 utilizationB,
    uint16 borrowingRateB,
    uint16 maxBorrowingRate
) internal {
    // (0%, 0%) -> (utilizationA, borrowingRateA) -> (utilizationB, borrowingRateB)
↪   -> (100%, maxBorrowingRate)
    reserve.borrowingRateConfig.utilizationA = uint128(
        Precision.FACTOR1E18.mul(utilizationA).div(Constants.PERCENT_100)
    );

    reserve.borrowingRateConfig.borrowingRateA = uint128(
        Precision.FACTOR1E18.mul(borrowingRateA).div(Constants.PERCENT_100)
    );
```

```
    reserve.borrowingRateConfig.utilizationB = uint128(
        Precision.FACTOR1E18.mul(utilizationB).div(Constants.PERCENT_100)
    );
    reserve.borrowingRateConfig.borrowingRateB = uint128(
        Precision.FACTOR1E18.mul(borrowingRateB).div(Constants.PERCENT_100)
    );
    reserve.borrowingRateConfig.maxBorrowingRate = uint128(
        Precision.FACTOR1E18.mul(maxBorrowingRate).div(
            Constants.PERCENT_100
        )
    );
}
```

Link to code

As we can see, this pretty much means that **before** the call of this function the current `utilisationRate` and the corresponding `currentBorrowingRate` **but after** the call of this function the current `utilisationRate` to give different `borrowingRate`. However, this function fails to `updateState` and `updateInterestRates` meaning that the new `borrowingRate` that the protocol wants to have for the current `utilisationRate` is **indefinite** when it will be set and for the period between now until the next action that will update the state, the old `borrowingRate` will be used.

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

1. Reserve is deployed with a specific `borrowingRateConfig`
2. After some time, protocol wants to update this `borrowingRateConfig` and calls `LendingPool::setBorrowingRateConfig`.
3. However, this new `borrowingRateConfig` will not be actually used until an action on the `LendingPool` being performed.

Impact

The impact of this vulnerability is that the protocol will actually have not control of **when** the new `borrowingRateConfig` will be **actually** activated. For the meantime between the `setBorrowingConfigRate` until the next action on the `LendingPool` which will update the

interest rates (which is ** indefinite**), the `currentBorrowingRate` that will be used will be the **old** one which was based on the previous `borrowingRateConfig`. This means that the protocol can lose funds since they will want for example for the current utilization rate to have 25% interest rate but instead they will have the previous 15% interest rate until someone update the state by `deposit/borrow`...

PoC

N/A

Mitigation

Consider calling `updateState` before the change of the configuration and `updateInterestRates` after the change of the configuration **during** the `LendingPool::setBorrowingRateConfig` call, in order to the update of the configuration to have instant and **immediate** effect on the `reserve.currentBorrowingRate`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/ExtraFi/extra-contracts/pull/1

Issue M-2: The reserve fee is calculated incorrectly leading to loss of fees for the reserve

Source: https://github.com/sherlock-audit/2024-11-extra-finance-v1-judging/issues/47

The protocol has acknowledged this issue.

Found by

A2-security, TessKimy, ck, jennifer37, zarkk01

Summary

The reserve fee is calculated incorrectly leading to loss of fees for the reserve. The actual minted shares will represent a lower value than the set `reserveFeeRate` represents.

Root Cause

In `ReserveLogic._mintToTreasury` the calculation of shares uses `reserveToETokenExchangeRate(reserve)` to calculate the shares represented by the set `reserveFeeRate`

https://github.com/sherlock-audit/2024-11-extra-finance-v1/blob/main/extra-contracts/contracts/libraries/logic/ReserveLogic.sol#L217-L246

The issue is that minting of shares will increase the total supply of eTokens and therefore lead to the shares minted actually representing a lower value of underlying that was intended.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

Lets consider the following values and use them in the `_mintToTreasury` function:

currentDebt = `2000e18` previousDebt = `1000e18` totalDebtAccrued = currentDebt.sub(previousDebt) = `2000e18-1000e18=1000e18` reserveValueAccrued = totalDebtAccrued.mul(feeRate).div(Constants.PERCENT_100 = `(1000e18*1500)/10000=150e18`

totalLiquidity = `10000e18` totalETokens = `5000e18` exchangeRate = reserveToETokenExchangeRate(reserve) = `(5000e18*1e18)/10000e18=0.5e18` feeInEToken = reserveValueAccrued.mul(exchangeRate).div(Precision.FACTOR1E18 = `(150e18*0.5e18)/1e18=75e18`

This means that the treasury will get minted `75e18` shares which are meant to represent `150e18` of the underlying. This is however flawed because after minting, the `totalETokens` now become `5000e18+75e18=5075e18` after the minted shares are added.

If the treasury shares are claimed, the exchange rate would be:

totalLiquidity.mul(Precision.FACTOR1E18).div(totalETokens) = `(10000e18*1e18)/5075e18`

and the value of underlying would therefore be:

underlyingTokenAmount = reserve.eTokenToReserveExchangeRate().mul(eTokenAmount).div(Precision.FACTOR1E18) = `(((10000e18*1e18)/5075e18)*75e18)/1e18=147.78e18`

This value is lower that the intended `150e18`

Impact

The protocol loses some of the reserve fees every time an operation is done. This figure will progressively compound over time as more transactions are done.

PoC

No response

Mitigation

The shares to be minted by the treasury should be:

```
reserveValueAccrued.mul(totalETokens).div(totalLiquidity - reserveValueAccrued))
```

Applying this to our case scenario above, the treasury would get minted approximately:

`150e18.mul(5000e18).div(10000e18-150e18)=76.14e18`

Redeeming these shares would therefore result in:

`(((10000e18*1e18)/5076.14e18)*76.14e18)/1e18=149.99e18`

Which is now correct.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.