HASH SECURITY

Ender Labs Security Review

Auditors

hash

1 Executive Summary

Over the course of 5 days in total, Ender Labs engaged with hash to review Ender Protocol V1. In this period of time a total of 28 issues were found.

Summary

Project Name	Ender Labs
Code Under Review	398531a3eb...
Type of Project	Yield, Restaking
Audit Timeline	10 May 2024 - 15 May 2024

Issues Found

High Risk	11
Medium Risk	5
Low Risk	5
Informational	7
Total Issues	28

2 Findings

2.1 High

2.1.1 Withdrawing POL is done without accounting for the current epoch

Although `epochStakingReward` (function which brings the contract to the latest state ie. mints end tokens based on rewards etc.) is invoked before the user's deposit and withdraw function, it is not invoked when withdrawing POL. This can cause an inflated withdrawablePOLAmount and hence remove more than what should be allowed

withdrawAvailablePOL

Recommendation

Invoke `epochStakingReward` inside the `withdrawAvailablePOL` function to bring the contract to the latest state

Status Fixed

2.1.2 Updating totalDepositInStrategy with withdrawing POL can cause further withdrawals to revert

When withdrawing POL from the treasury, the `totalDepositInStrategy` variable is reduced by the `_withdrawAmt`

```
    function withdrawAvailablePOL(address asset, uint256 amount) external onlyRole(DEFAULT_ADMIN_ROLE) {
        uint256 availableAmt = getWithdrawAmountForPOL();
        if (availableAmt < amount) revert InsufficientAmount();
        if (priorityStrategy == instadapp) {
            if (asset != stEth) revert InvalidAddress();
        }


        uint256 balanceTreasury = IERC20(asset).balanceOf(address(this));

        if (balanceTreasury >= amount) {
            _transferFunds(msg.sender, asset, amount);
        } else {
            if (balanceTreasury > 0) {
                _transferFunds(msg.sender, asset, balanceTreasury);
            }
=>          withdrawFromStrategy(msg.sender, asset, priorityStrategy, amount - balanceTreasury);
        }


        totalPOLdepositAmount += amount;
    }
```

```
    function withdrawFromStrategy(
        address _to,
        address _asset,
        address _strategy,
        uint256 _withdrawAmt
    ) internal validStrategy(_strategy) returns (uint256 _returnAmount) {

        ......

        if (totalDepositInStrategy < _withdrawAmt) revert InsufficientAmount();
        totalDepositInStrategy -= _withdrawAmt;

        emit StrategyWithdraw(_asset, _strategy, _withdrawAmt);
```

This can underflow as the `totalDepositInStrategy` variable keeps track of the `deposits` and doesn't include the `treasury returns` or the `excess gained by treasury using nominalYield` both of which contribute to withdrawable POL. Underflowing can make either the user's unable to withdraw their deposits or the protocol not able to withdraw the POL.

Recommendation

Avoid using the `withdrawFromStrategy` function to withdraw the POL

Status Fixed

2.1.3 Minted end token cannot be withdrawn out of the treasury

Unlike the legacy function `getMintedEnd()` which transfers the minted end token to admin, the `treasuryMint` function mints the endTokens to treasury itself

link

```
    function treasuryMint() external onlyRole(DEFAULT_ADMIN_ROLE) {

        ....

        IEndToken(endToken).mint(address(this), mintAmount);
        totalAddedMintAmount += mintAmount;
    }
```

But there is no functionality to withdraw the minted end tokens from the treasury causing the minted end tokens to remain in the treasury forever

Recommendation

Add a function to withdraw end tokens

Status Fixed

2.1.4 Updating fundsInfo with withdrawing POL can cause underflow / inflated deposit returns

When withdrawing POL from the treasury, the `fundsInfo` mapping is reduced by the `_withdrawAmt`

```
    function withdrawAvailablePOL(address asset, uint256 amount) external onlyRole(DEFAULT_ADMIN_ROLE) {
        uint256 availableAmt = getWithdrawAmountForPOL();
        if (availableAmt < amount) revert InsufficientAmount();
        if (priorityStrategy == instadapp) {
            if (asset != stEth) revert InvalidAddress();
        }


        uint256 balanceTreasury = IERC20(asset).balanceOf(address(this));

        if (balanceTreasury >= amount) {
            _transferFunds(msg.sender, asset, amount);
        } else {
            if (balanceTreasury > 0) {
                _transferFunds(msg.sender, asset, balanceTreasury);
            }
=>          withdrawFromStrategy(msg.sender, asset, priorityStrategy, amount - balanceTreasury);
        }


        totalPOLdepositAmount += amount;
    }
    function withdrawFromStrategy(
        address _to,
        address _asset,
        address _strategy,
        uint256 _withdrawAmt
    ) internal validStrategy(_strategy) returns (uint256 _returnAmount) {
        if (_asset == address(0) || _strategy == address(0)) revert ZeroAddress();
        if (_withdrawAmt == 0) revert ZeroAmount();
=>      if (fundsInfo[_asset] < _withdrawAmt) revert InsufficientAmount();
        fundsInfo[_asset] -= _withdrawAmt;
```

This can underflow as the `fundsInfo` mapping keeps track of the `deposits(+ bondFee)` and doesn't include the `treasury returns` or the `excess gained by treasury using nominalYield` both of which contribute to withdrawable POL. Underflowing can make either the user's unable to withdraw their deposits or the protocol not able to withdraw the POL. It can also result in inflated deposit returns as the deposit return is calculated as `(totalReturn * deposits / fundsInfo)` and due to POL withdarawal fundsInfo can become less that deposits

POC

Add the following test inside `test/integration/EnderProtocol.test.ts` and run `yarn test:integration` It is asserted than although withdrawable amount is 0.7e18, the fundsInfo mapping stores 0 and hence the actual withdrawal will revert

```
    describe.only("Test Hash", () => {
        it("disallows withdrawing pol", async () => {

            const baseRate = 300;
            const nominalYieldRate = -1000; // 10%
            await enderBond.setBondYieldBaseRate(baseRate);
            await treasury.connect(owner).setNominalYield(nominalYieldRate);
            const user1Addr = await user1.getAddress();
            const signature = await signatureDigest(
                user1,
                "bondContract",
                await enderBond.getAddress(),
                user1,
            );
            const userSign = {
                user: user1Addr,
                key: "0",
                signature,
            };
            const stEthAddr = await stEth.getAddress();
            const principal = ethers.parseEther("100");
            await stEth.connect(user1).submit(user1.address, { value: principal });
            await stEth.connect(user1).approve(await enderBond.getAddress(), principal);
            const maturity = 365;
            const bondFee = 0;

            await expect(
                enderBond
                    .connect(user1)
                    .deposit(
                        user1Addr,
                        principal,
                        maturity,
                        bondFee,
                        await stEth.getAddress(),
                        userSign,
                    ),
            ).to.emit(enderBond, "Deposit");

            // await treasury.depositInStrategy(stEthAddr, instadappLiteAddr, principal);
            const depositTime = await time.latest();
            await time.increase(maturity * 600);
            const currentTime = await time.latest();
            const actualDuration = currentTime - depositTime;

            await time.setNextBlockTimestamp(currentTime);
            const tx = await enderBond.connect(user1).withdraw(1);

            await time.setNextBlockTimestamp(currentTime);

            // now 10% ie. 0.7 tokens should be withdrawable for pol
            const withdrawableAmount = await treasury.getWithdrawAmountForPOL();
            expect(withdrawableAmount == 700000000000000000n).to.be.eq(true);
            // but withdrawing will revert due to fundsInfo being 0
            const fundsInfo = await treasury.fundsInfo(stEth);
            expect(fundsInfo == 0n).to.be.eq(true);

            console.log("withdrawble amount",withdrawableAmount);
            console.log("fundsInfo",fundsInfo);

            await expect(treasury.withdrawAvailablePOL(stEth,withdrawableAmount)).to.be.revertedWithCus
↪   tomError(treasury,"InsufficientAmount");
        })
    })
```

Recommendation

Only a part of the fundsInfo (bondFee) would be used when withdrawing the POL and the rest would be coming from other sources (treasury return, nominalYield). Hence another mechanism to account for how much of principal amount is present in the strategy and how much of it is being withdrawn would have to be used

Status The revert / inflated return issue is fixed. But the mechanism is still flawed. See #H-1 in update report

2.1.5 Unminted endToken amount associated with bond returns are not subtracted from tvl for available amount calculation

The available backing to mint new end tokens is calculated on a high level as `treasury tvl - current end token supply`. Hence the contract will mint upto `treasury tvl - current end token supply` equivalent of end tokens. But this is flawed as the `current end token supply` doesn't include the end tokens associated with bond returns which will only be minted when the bond is withdrawn. Hence this can decrease the backing % when the bonds are eventually withdrawn breaking the peg. Same issue occurs inside the `getWithdrawAmountForPOL` function

POC First apply the following fix for another existing issue (issue #5):

```
diff --git a/contracts/EnderTreasury.sol b/contracts/EnderTreasury.sol
index 5317ae5..edee3a9 100644
--- a/contracts/EnderTreasury.sol
+++ b/contracts/EnderTreasury.sol
@@ -223,10 +223,16 @@ contract EnderTreasury is Initializable, AccessControlUpgradeable, EnderELStrate
                 availableNominalAmount = 0;
             }
             int256 nominalYieldAmount = (depositReturn * nominalYield) / 10000;
-            if (nominalYieldAmount > 0  && uint256(nominalYieldAmount) > availableNominalAmount){
-                nominalYieldAmount = int256(availableNominalAmount);
-            }
+            // if (nominalYieldAmount > 0  && uint256(nominalYieldAmount) > availableNominalAmount){
+            //     nominalYieldAmount = int256(availableNominalAmount);
+            // }
             rebaseReward = (depositReturn + nominalYieldAmount - bondReturn);
+
+            // fix
+            if(rebaseReward > int256(availableNominalAmount) - bondReturn) {
+                rebaseReward = int256(availableNominalAmount) - bondReturn;
+            }
+
             address receiptToken = instadapp;
             instaDappLastValuation = IInstadappLiteV2(instadapp).convertToAssets(
                 IERC20(receiptToken).balanceOf(address(this))
```

Add the following test inside `test/integration/EnderProtocol.test.ts` and run `yarn test:integration`

```
    describe.only("Test Hash", () => {
        it("mints end token > backing even with fix due to not included end tokens associated with bond
↪   return", async () => {
            // since the endtokens associated with bond amount is not minted/considered in the
↪   available total supply, it is possible to mint more end
            const baseRate = 300;
            await enderBond.setBondYieldBaseRate(baseRate);

            const user1Addr = await user1.getAddress();
            const signature = await signatureDigest(
                user1,
                "bondContract",
                await enderBond.getAddress(),
                user1,
            );
            const userSign = {
                user: user1Addr,
                key: "0",
                signature,
            };
            const stEthAddr = await stEth.getAddress();
            const principal = ethers.parseEther("100");
            await stEth.connect(user1).submit(user1.address, { value: principal });
            await stEth.connect(user1).approve(await enderBond.getAddress(), principal);
            const maturity = 365;
            const bondFee = 0;

            await expect(
                enderBond
                    .connect(user1)
                    .deposit(
                        user1Addr,
                        principal,
                    maturity,
                    bondFee,
                    await stEth.getAddress(),
                    userSign,
                ),
            ).to.emit(enderBond, "Deposit");

            // await treasury.depositInStrategy(stEthAddr, instadappLiteAddr, principal);
            const depositTime = await time.latest();
            const halfMaturity = maturity / 2;

            await time.increase(halfMaturity * 600);
            let currentTime = await time.latest();

            await time.setNextBlockTimestamp(currentTime);

            let nominalYieldRate = 0; // 0%
            await treasury.connect(owner).setNominalYield(nominalYieldRate);

            await time.increase(halfMaturity * 600);

            currentTime = await time.latest();
            await time.setNextBlockTimestamp(currentTime);

            nominalYieldRate = 1000; // 10%
            await treasury.connect(owner).setNominalYield(nominalYieldRate);

            const tx = await enderBond.connect(user1).withdraw(1);

            const totalSupply = await endToken.totalSupply();
            const tvl = await treasury.getTreasuryTVL();

            console.log("totalSupply:", totalSupply);
            console.log("tvl in treasury", tvl);
            expect(tvl < (totalSupply /1000n)).to.be.eq(true);
        })
    }
```

It is asserted that treasury tvl is < (totalSupply /1000) ie. (end token total supply = 73500000000000000000 and treasury tvl = 70000000000000000)

Recommendation

Include the endToken amount associated with the unminted bond returns

Status Fixed

2.1.6 ETH deposits cannot be withdrawn due to withdrawal checks

ETH can be deposited in the bond and is identified in the protocol as address(0)

link

```
    function deposit(
        address user,
        uint256 principal,
        uint256 maturity,
        uint256 bondFee,
        address token,
        signData memory userSign
    ) public payable nonReentrant depositEnabled bondPaused returns (uint256 tokenId) {

        .....

        if (token == address(0)) {
            if (msg.value != principal) revert InvalidAmount();
            (bool suc, ) = payable(lido).call{value: msg.value}(
                abi.encodeWithSignature("submit(address)", address(this))
            );
            require(suc, "lido eth deposit failed");
            stEthFromDeposit = IERC20(stEth).balanceOf(address(this));
            IERC20(stEth).safeTransfer(address(endTreasury), stEthFromDeposit);
```

But the `withdrawFromStrategy` function will revert in case the asset is address(0)

link

```
    function withdrawFromStrategy(
        address _to,
        address _asset,
        address _strategy,
        uint256 _withdrawAmt
    ) internal validStrategy(_strategy) returns (uint256 _returnAmount) {
        if (_asset == address(0) || _strategy == address(0)) revert ZeroAddress();
```

This function will be called whenever user's attempt to withdraw their deposited assets and hence user's who deposited in ETH will not be able to withdraw

POC

Apply the following diff and run `yarn test:integration`. It is asserted that withdrawals in ETH will revert

```
diff --git a/test/integration/EnderProtocol.test.ts b/test/integration/EnderProtocol.test.ts
index 3adca8c..b80dfe6 100644
--- a/test/integration/EnderProtocol.test.ts
+++ b/test/integration/EnderProtocol.test.ts
@@ -47,10 +47,10 @@ describe("Ender Protocol", async () => {
     // Deploy contracts and set initial state before each test
     beforeEach(async () => {
         // This will reset the Hardhat Network before each test
-        await network.provider.request({
-            method: "hardhat_reset",
-            params: [],
-        });
+        // await network.provider.request({
+        //     method: "hardhat_reset",
+        //     params: [],
+        // });
         const accounts = await ethers.getSigners();

         owner = accounts[0];
@@ -175,6 +175,61 @@ describe("Ender Protocol", async () => {
         });
     });

+    describe.only("Test Hash", () => {
+        it("ETH deposits cannot be withdrawn", async () => {
+            // at the time of withdrawals, eth deposits will revert
+            const baseRate = 300;
+            const nominalYieldRate = 1000; // 10%
+            await enderBond.setBondYieldBaseRate(baseRate);
+            await treasury.connect(owner).setNominalYield(nominalYieldRate);
+            const user1Addr = await user1.getAddress();
+            const signature = await signatureDigest(
+                user1,
+                "bondContract",
+                await enderBond.getAddress(),
+                user1,
+            );
+            const userSign = {
+                user: user1Addr,
+                key: "0",
+                signature,
+            };
+            const stEthAddr = await stEth.getAddress();
+            const principal = ethers.parseEther("100");
+            // await stEth.connect(user1).submit(user1.address, { value: principal });
+            // await stEth.connect(user1).approve(await enderBond.getAddress(), principal);
+            const maturity = 365;
+            const bondFee = 0;
+
+            await expect(
+                enderBond
+                    .connect(user1)
+                    .deposit(
+                        user1Addr,
+                        principal,
+                        maturity,
+                        bondFee,
+                        ethers.ZeroAddress,
+                        userSign,
+                    {value:principal}),
+            ).to.emit(enderBond, "Deposit");
+            console.log("deposited");
+            // await treasury.depositInStrategy(stEthAddr, instadappLiteAddr, principal);
+            const depositTime = await time.latest();
+            await time.increase(maturity * 600);
+            const currentTime = await time.latest();
+            const actualDuration = currentTime - depositTime;
+
+            await time.setNextBlockTimestamp(currentTime);
+            // await expect(lock.withdraw()).to.be.revertedWith("You can't withdraw yet");
+            // const tx = await enderBond.connect(user1).withdraw(1);
+            await
↪  expect(enderBond.connect(user1).withdraw(1)).to.be.revertedWithCustomError(treasury,"ZeroAddress");
+
+
+
+        })
+    })
+
     describe("Multi users", () => {
         it("multi bonds(3 users)", async () => {
             const [, , , customer1, customer2, customer3] = await ethers.getSigners();
diff --git a/test/utils/deployer.ts b/test/utils/deployer.ts
```

```
index 8e51b95..80211e2 100644
--- a/test/utils/deployer.ts
+++ b/test/utils/deployer.ts
@@ -58,7 +58,8 @@ export class Deployer {
         const enderStakeEth = await this.enderStakeEth();
         const endTokenAddr = await endToken.getAddress();
         const sEndTokenAddr = await sEndToken.getAddress();
-        const stEthAddr = await stEth.getAddress();
+        // const stEthAddr = await stEth.getAddress();
+        const stEthAddr = process.env.LIDO_SEPOLIA_ADDR;
         const enderStakeEthAddr = await enderStakeEth.getAddress();
         // const lidoAddr = await lido.getAddress();
         const lidoAddr = process.env.LIDO_SEPOLIA_ADDR;
```

Recommendation Since steth cannot be returned back as eth, avoid this feature

Status Fixed

2.1.7 Burn and mint functions are not access controlled

The burn functionality of `Ender Stake Ether` and burn and mint functionalities of `sEndToken` token is not access controlled

link

```
    function burn(address from, uint256 amount) public {
        _burn(from, amount);
    }
```

link

```
    function burn(address from, uint256 value) public {
        super._transfer(from, address(0xdead), value);
    }


    function whitelist(address _whitelistingAddress, bool _action) external
 ↪  onlyRole(DEFAULT_ADMIN_ROLE) {
        isWhitelisted[_whitelistingAddress] = _action;
        emit WhitelistChanged(_whitelistingAddress, _action);
    }


    ///for testing purpose
    function mint(address to, uint256 amount) public {
        _mint(to, amount);
    }
```

Hence any user can burn the tokens of any other user or mint tokens freely

Recommendation

Introduce proper access controls

Status Fixed

2.1.8 Double counting of `totalRewardFeeAmount` in treasury tvl

The treasury tvl calculation includes `IInstadappLiteV2(receiptToken).convertToAssets(receiptTokenAmount) + totalRewardFeeAmount ....`

link

```
    function getTreasuryTVL() public view returns (uint256 treasuryTVL) {
        address receiptToken = instadapp;
        uint256 receiptTokenAmount = IInstadappLiteV2(receiptToken).balanceOf(address(this));
        uint256 totalDepositReturn = IEnderBond(enderBond).totalDepositReturn();
        treasuryTVL = IInstadappLiteV2(receiptToken).convertToAssets(receiptTokenAmount) +
↪   totalRewardFeeAmount + totalPOLdepositAmount + IERC20(stEth).balanceOf(address(this)) -
↪   totalDepositReturn;
    }
```

`totalRewardFeeAmount` will be deposited inside the strategy and hence will be included inside `IInstadappLiteV2(receiptToken).convertToAssets(receiptTokenAmount)`. This will cause the double counting of totalRewardFeeAmount leading to an inflated treasury tvl

Recommendation

Remove `totalRewardFeeAmount` from the calculation

Status Fixed

2.1.9 Incorrect check with nominalYieldAmount is made instead of the entire mint amount

To keep the end supply fully backed, it should be ensured that the minted end tokens can at max be equal to (treasury tvl * 1000)

link

```
    function stakeRebasingReward(address _tokenAddress) external onlyStaking returns (int256
↪   rebaseReward) {

        ....

        } else {
            depositReturn = depositReturn * 1000;
            totalEndSupply += uint256(depositReturn);
            uint256 treasuryTVL = getTreasuryTVL();
            uint256 totalEndAmount = IERC20(endToken).totalSupply();
            uint256 availableNominalAmount = (treasuryTVL - totalRewardFeeAmount) * 1000;
            if (availableNominalAmount > totalEndAmount) {
                availableNominalAmount -= totalEndAmount;
            } else {
                availableNominalAmount = 0;
            }
            int256 nominalYieldAmount = (depositReturn * nominalYield) / 10000;
=>          if (nominalYieldAmount > 0  && uint256(nominalYieldAmount) > availableNominalAmount){
                nominalYieldAmount = int256(availableNominalAmount);
            }
            rebaseReward = (depositReturn + nominalYieldAmount - bondReturn);
```

Here instead of comparing the entire endToken amount that is going to be minted (ie. depositReturn + nominalYieldAmount + bondReturn), the nominalYieldAmount is compared with the available treasury value. Hence the check is ineffective and allows minting of end tokens without enough backing

POC

Add the following test inside `test/integration/EnderProtocol.test.ts` and run `yarn test:integration` It is asserted than although treasury tvl is only 7e18 stETH, 7.7ke18 end token is minted if nominal rate is set to 10%

```
    describe.only("Test Hash", () => {
        it("mints end token > backing", async () => {
            // normal without fees deposit. then if there is a nominal positive yield value kept then
↪   the minted end will be greater than the backing and the checks in the contract doesn't cover this
            const baseRate = 300;
            const nominalYieldRate = 1000; // 10%
            await enderBond.setBondYieldBaseRate(baseRate);
            await treasury.connect(owner).setNominalYield(nominalYieldRate);
            const user1Addr = await user1.getAddress();
            const signature = await signatureDigest(
                user1,
                "bondContract",
                await enderBond.getAddress(),
                user1,
            );
            const userSign = {
                user: user1Addr,
                key: "0",
                signature,
            };
            const stEthAddr = await stEth.getAddress();
            const principal = ethers.parseEther("100");
            await stEth.connect(user1).submit(user1.address, { value: principal });
            await stEth.connect(user1).approve(await enderBond.getAddress(), principal);
            const maturity = 365;
            const bondFee = 0;

            await expect(
                enderBond
                    .connect(user1)
                    .deposit(
                        user1Addr,
                        principal,
                        maturity,
                        bondFee,
                        await stEth.getAddress(),
                        userSign,
                    ),
            ).to.emit(enderBond, "Deposit");

            // await treasury.depositInStrategy(stEthAddr, instadappLiteAddr, principal);
            const depositTime = await time.latest();
            await time.increase(maturity * 600);
            const currentTime = await time.latest();
            const actualDuration = currentTime - depositTime;

            await time.setNextBlockTimestamp(currentTime);
            const tx = await enderBond.connect(user1).withdraw(1);

            await time.setNextBlockTimestamp(currentTime);
            const totalSupply = await endToken.totalSupply();
            const tvl = await treasury.getTreasuryTVL();

            console.log("totalSupply:", totalSupply);
            console.log("tvl in treasury", tvl);
            expect(tvl < (totalSupply /1000n) ).to.be.eq(true);
        })
    })
```

Recommendation

Compare with the full amount instead of just the nominalYieldAmount ie.

```
            rebaseReward = (depositReturn + nominalYieldAmount - bondReturn);
            if(rebaseReward > int256(availableNominalAmount) - bondReturn) {
                rebaseReward = int256(availableNominalAmount) - bondReturn;
            }
```

Status Fixed

2.1.10 sEnd transfers instead of burn

The `burn` function of `sEnd` token transfer the tokens instead of burning. This will cause the totalSupply to remain as is and hence corrupt all the send:end ratio calculation

link

```
    function burn(address from, uint256 value) public {
        super._transfer(from, address(0xdead), value);
    }
```

Recommendation

Use the `_burn` function itself instead of transferring

Status Fixed

2.1.11 depositInStrategy doesn't increase the fundsInfo

depositInStrategy is also used to deposit funds externally (rather than via bonds). Such funds and its returns should belong to the treasury rather than the bond depositor's. It should also be withdrawable in future (for POL). Both these conditions require updating the deposited amount in the fundsInfo mapping as it is decremented at the time of withdrawal and is also used to obtain the ratio of `bond deposits:total deposits`. But the `depositInStrategy` function doesn't update this

link

```
    function depositInStrategy(address _asset, address _strategy, uint256 _depositAmt) public
↪   validStrategy(_strategy) {
        if (_depositAmt == 0) revert ZeroAmount();
        if (_asset == address(0) || _strategy == address(0)) revert ZeroAddress();
        if (_strategy == instadapp) {
            IERC20(_asset).approve(_strategy, _depositAmt);
            IInstadappLiteV2(instadapp).deposit(_depositAmt, address(this)); // note for testing we
↪   changed the function sig.
            instaDappDepositValuations += _depositAmt;
        }
        // else if (_strategy == lybraFinance) {
        //     IERC20(_asset).approve(lybraFinance, _depositAmt);
        //     ILybraFinance(lybraFinance).depositAssetToMint(_depositAmt, 0);
        // } else if (_strategy == eigenLayer) {
        //     //Todo will add the instance while going on mainnet.
        // }
        totalDepositInStrategy += _depositAmt;
        emit StrategyDeposit(_asset, _strategy, _depositAmt);
    }
```

Recommendation

Update the `fundsInfo` mapping when depositing externally via `depositInStrategy`

Status Fixed

2.2 Medium

2.2.1 Inflation attack on staking contract is possible

There is no protection against the share value inflation attack in the staking contract

stake function

The rebase reward would be transferred to the staking contract and the first staker as can stake as little as 1 wei to value the share at this amount. The attacker can inflate the share value by donating the to the staking contract and forcing the recalculation of the `rebaseIndex` by donating 1 wei in instadapp (this will make the reward amount > 0 and hence recalculate the rebase index)

forcing rebase index calculation

```
        if (totalReward > 0) {
            rebaseRefractionReward = (uint256(totalReward) * bondRewardPercentage) / 10000;
            uint256 rebaseEndAmount = uint256(totalReward) - rebaseRefractionReward;
            rebaseEndAmountPerDay = rebaseEndAmount * SECONDS_IN_DAY / (block.timestamp -
↪   latestRebaseUpdateTime);
            if (ISEndToken(endToken).totalSupply() > 0) {
                stateStakingYield = rebaseEndAmountPerDay * 10000 / ISEndToken(endToken).totalSupply();
            } else {
                stateStakingYield = 10000;
            }
            ISEndToken(endToken).mint(address(this), rebaseEndAmount);
            calculateRebaseIndex();
```

Recommendation

Keep a check to ensure that the minted shares is non-zero when staking via the mint function

Status See update report

2.2.2 Migration can cause double counting of the migrating value

When migrating, the funds are moved externally to the treasury contract before the `epochStakingReward` function is called

```
    function migrateBondFromLiquidityDeposit(uint lastIndex) external onlyRole(DEFAULT_ADMIN_ROLE)
↪   notMigratedDepositLiquidity {
        for (uint256 i = 1; i <= lastIndex; i++) {
            IEndToken(endToken).distributeRefractionFees();
            endStaking.epochStakingReward(stEth);
            (address user, uint256 principal, uint256 bondFees, uint256 maturity) =
                depositContract.depositedIntoBond(i);
            _deposit(user, principal, maturity, stEth, bondFees * 100);
        }
        isMigratedDepositLiquidity = true;
    }
```

Since the treasury values `stETH.balanceOf(treasury)` in its tvl, it will also count the steth which is part of migration as its tvl. This can cause overminting of the end token as rebase reward incase of a positive nominal yield

Eg: actual tvl = 100, deposit return 100, bond return 10, nominal yield 20. with the actual tvl, only 10 will be minted 90 end tokens could be minted even though `deposit return + nominal yield - bond return` is 110. but if the migration had more than 100 steth tokens, now tvl will be calculated as 200 and hence 110 end tokens will be minted as rebase reward

Recommendation

Invoke `epochStakingReward` before the funds are moved to the treasury

Status Fixed

2.2.3 Usage of rebase index to determine the sEnd:end ratio is incorrect

`rebaseIndex` is used to determine the ratio of send:end when burning/minting sEnd tokens. The current implementation caches a `rebaseIndex` value and performs mints/burns of `sEnd` based on that instead of the latest ratio

```
    function stake(uint256 amount, signData memory userSign) external stakingEnabled
↪   stakingContractPaused {
        if (amount == 0) revert InvalidAmount();
        if (isWhitelisted) {
            address signAddress = _verify(userSign);
            if (signAddress != contractSigner || userSign.user != msg.sender) revert NotWhitelisted();
        }


        _epochStakingReward(stEth);


        if (ISEndToken(endToken).balanceOf(address(this)) == 0) {
=>          uint256 sEndAmount = calculateSEndTokens(amount);
```

```
    function calculateSEndTokens(uint256 _endAmount) public view returns (uint256 sEndTokens) {
        if (rebasingIndex == 0) {
            sEndTokens = _endAmount;
        } else {
            sEndTokens = ((_endAmount * 1e18) / rebasingIndex);
        }
    }
```

If the rebase index was always updated whenever necessary it would be fine with the effect that the excess deposited endTokens could be shared with newer stakers too instead of belonging solely to the previous staker's and a possible abrupt change in the sEnd token valuation. But the rebase index is not always updated correctly

In `_epochStakingReward`, the rebase index is calculated before the `send` tokens associated with fees are mint and is not recalculated after

```
    function _epochStakingReward(address _asset) internal {

            ....

            ISEndToken(endToken).mint(address(this), rebaseEndAmount);
=>          calculateRebaseIndex();


            uint256 sendTokens = calculateSEndTokens(rebaseRefractionReward);
            totalRebaseFeeAmount += sendTokens;
            ISEndToken(sEndToken).mint(enderBond, sendTokens);
            IEnderBond(enderBond).epochRebasingFeeShareIndex(sendTokens);
            ISEndToken(endToken).mint(address(this), rebaseRefractionReward);
```

Hence in the first iteration where all the rebase reward should be belonging to the bond depositors if there was `rebaseRefractionReward`(because there can be no other end staked), the `rebaseIndex` would still be cached as 1e18 even though the live ratio of send:end would be > 1e18. If any other depositor makes a deposit before any further deposit return is made, then they will also be able to share this rebaseReward. If any deposit return is made, then the rebaseReward will solely belong to the firstDepositor causing a clear inconsistency

Recommendation

The usage of rebaseIndex is not necessary and it simply creates additional complexity. Hence the rebaseIndex can be avoided completely to implement the usual way of valuing tokens (taking currently supply, current balance). Else the `rebaseIndex` will have to be updated after every change in the send/end balance to avoid inconsistencies. Currently it is not done in `_epochStakingReward`,`unstake` and `stake`

Status Fixed

2.2.4 Refraction fees will be lost incase there is no active deposit

In case the `totalRefractionPrincipal` is 0, the `tradingFeeShareIndex` will not be updated. This is correct.

link

```
    function epochTradingFeeShareIndex(uint256 _reward) external onlyEndToken {
        if (totalRefractionPrincipal > 0) {
            IERC20(endToken).safeTransferFrom(msg.sender, address(this), _reward);
            tradingFeeShareIndex += (_reward * 10 ** 18) / totalRefractionPrincipal;
        }
        emit TradingFeeShareIndexUpdated(tradingFeeShareIndex);
    }
```

But the collected fees is still cleared from the endToken contract causing this fees to be lost forever

link

```
    function distributeRefractionFees() external onlyRole(ENDERBOND_ROLE) {
        // if (lastEpoch + 1 days > block.timestamp) revert InvalidEarlyEpoch();
        uint256 feesToTransfer = refractionFeeTotal;
        if (feesToTransfer != 0) {
=>          refractionFeeTotal = 0;
            lastEpoch = block.timestamp;
            _approve(address(this), enderBond, feesToTransfer);
        }
        IEnderBond(enderBond).epochTradingFeeShareIndex(feesToTransfer);
        emit RefractionFeesDistributed(enderBond, feesToTransfer);
    }
```

Recommendation

Only set `refractionFeeTotal` to 0 incase refraction principal is non-zero

Status Fixed

2.2.5 Multiple fees cannot be set to 0

`bondRewardPercentage` and `refractionFeePercentage` cannot be set to 0 due to checks inside the respective function. This is not the intended functionality and the option to set the feePercentage to 0 should be available

link

```
    function setBondRewardPercentage(uint256 percent) external onlyRole(DEFAULT_ADMIN_ROLE) {
        if (percent == 0) revert InvalidAmount();


        bondRewardPercentage = percent;
        emit PercentUpdated(bondRewardPercentage);
    }
```

link

```
    function setFee(uint256 fee) public onlyRole(DEFAULT_ADMIN_ROLE) {
        if (fee == 0) revert InvalidParam();


        refractionFeePercentage = fee;


        emit FeeUpdated(fee);
    }
```

Recommendation

Remove the `fee == 0 then revert` checks

Status Fixed

2.3 Low

2.3.1 rebasingIndexPerSecond calculation is incorrect

When calculating the rebaseIndexPerSecond, the `sendTokens` associated with the rebase fees has already minted to the bond which increases the totalSupply

```
            ISEndToken(sEndToken).mint(enderBond, sendTokens);
            IEnderBond(enderBond).epochRebasingFeeShareIndex(sendTokens);
            ISEndToken(endToken).mint(address(this), rebaseRefractionReward);


            oldSEndBalance = ISEndToken(sEndToken).totalSupply();
            if (block.timestamp > latestRebaseUpdateTime && oldSEndBalance > 0) {
                rebasingIndexPerSecond =
                    (rebaseEndAmount * 1e18) /
                    (oldSEndBalance * (block.timestamp - latestRebaseUpdateTime));
            }
```

When calculating the rebasingIndexPerSecond, this will also get included without the inclusion of the associated rebaseRefractionReward leading to incorrect value for `rebasingIndexPerSecond`

Recommendation

Perform the calculation before the tokens are minted to the bond

Status Acknowledged

2.3.2 stateStakingYield calculation is incorrect / will return 0 after few deposits

stateStakingYield is calculated as follows:

```
if (ISEndToken(endToken).totalSupply() > 0) {
                stateStakingYield = rebaseEndAmountPerDay * 10000 / ISEndToken(endToken).totalSupply();
            } else {
                stateStakingYield = 10000;
            }
```

If the division with `(endToken).totalSupply()` is intended, the precision used ie. 10000, is low and can start returning 0 soon

POC: 1 yr net yield : 7% 100 eth is the deposit amount let there be 10 such deposits hence after 1 yr total end supply = 70 for a deposit of 100 eth, rebase amount per day = 2.5 / 365 (3% base yield, 1.5x multipler for 1 yr deposit, hence rebase reward % == 2.5) stateStakingYield = (2.5 / 365) * 10000 / 70 < 1 == 0

Recommendation

Increase the precision or if the usage of endToken is incorrect, update it

Status Acknowledged

2.3.3 Previous instadapp valuation will be used in case `totalReceiptBalance` is 0 causing potential underflow

In case the current balance in strategy is 0, the instaDappLastValuation is not updated

```
    function stakeRebasingReward(address _tokenAddress) external onlyStaking returns (int256
↪   rebaseReward) {
        int256 bondReturn = int256(IEnderBond(enderBond).calcEpochBondAmount(block.timestamp));
        int256 depositReturn = int256(calculateDepositReturn(_tokenAddress));
        balanceLastEpoch = IERC20(_tokenAddress).balanceOf(address(this));
        if (depositReturn == 0) {
            rebaseReward = 0;
            address receiptToken = instadapp;
=>          if (IInstadappLiteV2(receiptToken).balanceOf(address(this)) > 0) {
                instaDappLastValuation = IInstadappLiteV2(instadapp).convertToAssets(
                    IERC20(receiptToken).balanceOf(address(this))
                );
            }
            instaDappWithdrawlValuations = 0;
            instaDappDepositValuations = 0;
```

Although the balance will not become 0 in normal conditions, it can in rare scenarios such as when there is no endToken minted yet and a donation to the treasury happens followed by withdrawal of the entire amount as POL. In such a case the future calculation of `totalReturn` can revert due to underflow

```
    function calculateTotalReturn() internal view returns (uint256 totalReturn) {
        address receiptToken = instadapp;
        uint256 receiptTokenAmount = IInstadappLiteV2(receiptToken).balanceOf(address(this));
        if (receiptTokenAmount > 0) {
            totalReturn =
                IInstadappLiteV2(receiptToken).convertToAssets(receiptTokenAmount) +
                instaDappWithdrawlValuations -
                instaDappDepositValuations -
                instaDappLastValuation;
        }
    }
```

Recommendation

Set the valuation to 0 incase the token balance is 0

Status Fixed

2.3.4 No setter for `rateOfChange`

There is no setter available for the rate of change variable. Hence it would not be possible to change the value

variable link

Recommendation

Add a setter function

Status Fixed

2.3.5 getMintedEnd cannot be called by public

The function `getMintedEnd()` is open for public to call and it transers the minted end token to the admin

link

```
    function getMintedEnd() external {
        mintAndVest();

        ....
```

link

```
    function mintAndVest() internal {
        uint256 time = block.timestamp;
        if (time >= lastYear * 31536000 + 365 days) {
            uint256 mintAmount = (totalSupply() * mintFee) / 10000;
            yearlyVestAmount[time / 31536000] = VestAmount(mintAmount, true, true, true);
=>          mint(address(this), mintAmount);
```

link

```
    function mint(address to, uint256 amount) public onlyRole(MINTER_ROLE) {
        if (to == address(0)) revert ZeroAddress();


        _mint(to, amount);
    }
```

But since inside the `mintAndVest` function, `mint` is called instead of the internal `_mint`, this will disallow calls from addresses that don't have the minter role

Recommendation

Change to internal _mint

Status Fixed

2.4 Informational

2.4.1 `rebasingIndexPerSecond` calculation changes it meaning

`rebasingIndexPerSecond` is originally calculated as:

```
rebasingIndexPerSecond =
                    (rebaseEndAmount * 1e18) /
                    (oldSEndBalance * (block.timestamp - latestRebaseUpdateTime));
            }
```

Giving it a meaning of `rebase reward amount per sEnd token per second`

Whenever a user unstakes, it is updated as:

```
        uint256 sEndTotalSupply = ISEndToken(sEndToken).totalSupply();
        if (sEndTotalSupply == 0) {
            rebasingIndexPerSecond = 0;
        } else {
            rebasingIndexPerSecond = (rebasingIndexPerSecond * oldSEndBalance) / sEndTotalSupply;
```

This increases the `rebasingIndexPerSecond` and changes its meaning

Status Acknowledged

2.4.2 status is set twice at the time of initialization

`setStatus` sets the status variable which is again set in the very next line at the time of initialization

```
    function initialize() external initializer {
        __ERC20_init("sEndToken", "sEnd");
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        setStatus(1);
        status = 1;
        enableOrDisableTX = false;
    }
```

Recommendation

Only set once

Status Fixed

2.4.3 Testing values are still used

In multiple places the values used for testing are still present and even imports hardhat/console.sol

```
    function initialize(address endToken_, address _lido, address _oracle) public initializer {
        __Ownable_init();
        __EIP712_init("EnderBond", "1");
        rateOfChange = 100;
        lido = _lido;
        setAddress(endToken_, 2);
        // todo set the value according to doc
        minDepositAmount = 1000000000000000;
        txFees = 200;
        enderOracle = IEnderOracle(_oracle);
        bondYieldBaseRate = 100;
        SECONDS_IN_DAY = 600; // note for testing purpose we have set it to 10 mint
        interval = 10 * 60; // note for testing purpose we have set it to 10 mint
```

Recommendation

Update the values to correct ones before deployment

Status Acknowledged

2.4.4 Legacy code is present

The protocol wants of feature of extra minting of end tokens. Due to not removing legacy code, there are multiple functions currently present in the codebase to acheive this

getMintedEnd treasuryMint

Recommendation

Remove the legacy code

Status Fixed

2.4.5 Unused code fragments are kept in the code

There are multiple instances of unused variables/ functions that are present in the codebase:

enableOrDisableTX var unused https://github.com/enderprotocol/ender-v1/blob/audit/contracts/ERC20/SEndToken.sol#L10

collect function unused https://github.com/enderprotocol/ender-v1/blob/audit/contracts/EnderTreasury.sol#L326-L329

mintCount var unused https://github.com/enderprotocol/ender-v1/blob/audit/contracts/ERC20/EndToken.sol#L34

isSet var unused https://github.com/enderprotocol/ender-v1/blob/audit/contracts/EnderBond.sol#L63

strategyToReceiptToken https://github.com/enderprotocol/ender-v1/blob/audit/contracts/EnderTreasury.sol#L22

Recommendation

Remove the unused code fragments

Status Fixed

2.4.6 totalEndSupply is used instead of the token's current totalSupply for extra endToken minting

totalEndSupply is used instead of the token's current totalSupply for extra endToken minting link

```
    function treasuryMint() external onlyRole(DEFAULT_ADMIN_ROLE) {

        ....

=>      uint256 mintAmount = (totalEndSupply * mintRate) / 10000;
```

Recommendation

In case not intended, use token.totalSupply()

Status Fixed

2.4.7 Ender treasury inheriting ELS strategy

EnderTreasury is inheriting EnderELStrategy which is unwanted and an incorrect inheritance link

```
contract EnderTreasury is Initializable, AccessControlUpgradeable, EnderELStrategy {
```

Recommendation

Avoid inheriting EnderELStrategy

Status Fixed