Balmy Earn

Security Review

Cantina Managed review by:
Blockdev, Security Researcher
Ladboy233, Security Researcher

December 28, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Balmy provides a secure and intuitive environment for users to explore new financial opportunities.

From Nov 11th to Dec 15th the Cantina team conducted a review of earn-core and earn-periphery on commit hashes c71cfcee and 4a83e654 respectively. The team identified a total of **38** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 5
- Medium Risk: 9
- Low Risk: 10
- Gas Optimizations: 4
- Informational: 10

3 Findings

3.1 High Risk

3.1.1 `CompoundV2Connector#_convertSharesToAssets` magnitude scalar can be too large and truncate the asset worth

Severity: High Risk

Context: CompoundV2Connector.sol#L378-L383

Description: The formula to compute the cToken worth is:

```
uint256 underlyingTokenDecimals = Math.max(Token.NATIVE_TOKEN == asset ? 18 : ICERC20(asset).decimals(), 8);
uint256 magnitude = (10 + underlyingTokenDecimals);
return shares.mulDiv(cToken().exchangeRateStored(), 10 ** magnitude, rounding);
```

if the underlying token is ETH or DAI, which has 18 decimals, the magnitude will be $10 + 18 = 28$, and the asset worth becomes `cToken balance * exchange rate / 10 ** 28`, yet the true formula is `cToken balance * exchange rate / 10 ** 18`, then the worth asset estimation is off by 10^{10}.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

interface ICERC20 is IERC20 {
    function mint(uint256 underlyingAmount) external returns (uint256);
    function redeemUnderlying(uint256 underlyingAmount) external returns (uint256);
    function exchangeRateStored() external view returns (uint256);
    function decimals() external view returns (uint256);
    function getCash() external view returns (uint256);
    function totalReserves() external view returns (uint256);
    function totalBorrows() external view returns (uint256);
    function redeem(uint redeemTokens) external returns (uint);
    function exchangeRateCurrent() external returns (uint);
  }

  interface IComptroller {
    function claimComp(address[] memory holders, ICERC20[] memory cTokens, bool borrowers, bool suppliers)
    ↪  external;
    function compSpeeds(address cToken) external view returns (uint256);
    function compAccrued(address) external view returns (uint256);
    function mintGuardianPaused(ICERC20 cToken) external view returns (bool);
  }


contract CounterTest is Test {

    ICERC20 cToken = ICERC20(0x5d3a536E4D6DbD6114cc1Ead35777bAB948E3643); // cDAI
    IERC20 asset = IERC20(0x6B175474E89094C44Da98b954EedeAC495271d0F); // DAI
    IERC20 comp = IERC20(0xc00e94Cb662C3520282E6f5717214004A7f26888); // COMP

    address cTokenHolder = 0xB0b0F6F13A5158eB67724282F586a552E75b5728; // cDAI holder

    using stdStorage for StdStorage;
    StdStorage stdlib;

    function setUp() public {

    }

    function testRedeem() public {

        address user = 0x90581aFC83520a6493768521663B3df92153cEE20;

        vm.startPrank(user);

        uint256 balance = cToken.balanceOf(user);
```

```
        console.log("balance: %s", balance);

        uint256 decimals = ICERC20(address(asset)).decimals();

        uint256 magnitude = (10 + decimals);

        uint256 daiBefore = asset.balanceOf(user);

        uint256 expected_redeem_dai = balance * cToken.exchangeRateStored() / 10 ** magnitude;

        console.log("redeem dai with lagging interest: %s", expected_redeem_dai);

        uint256 accurate_redeem_dai = balance * cToken.exchangeRateCurrent() / 10 ** magnitude;

        console.log("accurate redeem dai             : %s", accurate_redeem_dai);

        cToken.redeem(balance);

        uint256 daiAfter = asset.balanceOf(user);

        console.log("redeemed dai balance change     : %s", daiAfter - daiBefore);

    }
}
```

Run the test with:

```
forge test -vvv --match-test "testRedeem" --fork-url "https://eth.llamarpc.com" --block-number 21182834
```

The output is:

```
Ran 1 test for test/Counter.t.sol:CounterTest
[PASS] testRedeem() (gas: 196442)
Logs:
  balance: 12014076301212601
  redeem dai with lagging interest: 286580759895293
  accurate redeem dai             : 286583666872423
  redeemed dai balance change     : 2865836668724232146371784
```

The computed redeemed DAI amount is 10 digit off by the redeemed dai balance change.

Recommendation: Use `10 ** 18` for the magnitude, as `cToken` has 8 decimals.

Balmy: We applied a fix for finding "CompoundV2Connector#_convertSharesToAssets does not consider pending cToken interest", which should also fix this. The fix is located in PR 101.

Cantina Managed: Fixed.

3.1.2 Complete loss event counter resets to 0 on partial loss

Severity: High Risk

Context: YieldMath.sol#L67-L83, YieldMath.sol#L137-L140

Description: `YieldMath.calculateAccum()` is supposed to calculate updated yield accumulate, loss accumulate and complete loss event counter. However, on a partial loss event, this function doesn't set `newStrategyCompleteLossEvents` which is the updated complete loss event counter. Thus, it is implicitly set to 0.

This effects yield and balance calculation since `YieldMath.calculateBalance()` compares number of recorded complete loss events for the position with the number of complete loss event for the strategy.

Proof of Concept:

```
function test_zero_completeLossEvents() public {
  uint256 amountToDeposit1 = 100_000;
  uint256 amountToBurn = 1000;
  uint256 amountToReward = amountToBurn;
  erc20.mint(address(this), type(uint256).max);
  INFTPermissions.PermissionSet[] memory permissions =
```

```
    PermissionUtils.buildPermissionSet(operator, PermissionUtils.permissions(vault.WITHDRAW_PERMISSION()));
  bytes memory misc = "1234";

  address[] memory strategyTokens = new address[](2);
  strategyTokens[0] = address(erc20);
  strategyTokens[1] = address(anotherErc20);
  (StrategyId strategyId, EarnStrategyStateBalanceMock strategy) =
    strategyRegistry.deployStateStrategy(strategyTokens);

  (uint256 positionId1,) =
    vault.createPosition(strategyId, address(erc20), amountToDeposit1, positionOwner, permissions,
    ↪  creationData, misc);
  uint256 losses;
  uint256[] memory balances1;

  for (uint256 i = 1; losses <= 3; i++) {
    vault.createPosition(strategyId, address(erc20), amountToDeposit1, positionOwner, permissions,
    ↪  creationData, misc);

    if (i % 2 == 0) {
      anotherErc20.burn(address(strategy), anotherErc20.balanceOf(address(strategy)));
      losses++;
    } else {
      anotherErc20.mint(address(strategy), amountToReward);
    }
  }
  (,,,YieldLossDataForToken[] memory y) = vault.getStrategyYieldData(strategyId);
  assertEq(y[0].completeLossEvents, 3);
  vault.createPosition(strategyId, address(erc20), amountToDeposit1, positionOwner, permissions, creationData,
  ↪  misc);
  anotherErc20.mint(address(strategy), amountToReward);
  (,,,y) = vault.getStrategyYieldData(strategyId);
  assertEq(y[0].completeLossEvents, 4);
  vault.createPosition(strategyId, address(erc20), amountToDeposit1, positionOwner, permissions, creationData,
  ↪  misc);
  anotherErc20.burn(address(strategy), anotherErc20.balanceOf(address(strategy))/2);
  vault.createPosition(strategyId, address(erc20), amountToDeposit1, positionOwner, permissions, creationData,
  ↪  misc);

  (,,,y) = vault.getStrategyYieldData(strategyId);
  assertEq(y[0].completeLossEvents, 0);
  vault.createPosition(strategyId, address(erc20), amountToDeposit1, positionOwner, permissions, creationData,
  ↪  misc);
  (,,,y) = vault.getStrategyYieldData(strategyId);
  assertEq(y[0].completeLossEvents, 0);
}
```

This shows that `completeLossEvents` moves from 3 to 4 to 0 which shouldn't be possible.

Recommendation: Set `newStrategyCompleteLossEvents` to `previousStrategyCompleteLossEvents` in the case of partial loss.

Balmy: Fixed in PR 74.

Cantina Managed: Fix looks good.

3.1.3 The hint is not fetched properly from lido withdrawal queue and make all lido delayed withdraw revert

Severity: High Risk

Context: LidoSTETHDelayedWithdrawalAdapter.sol#L173-L175

Description:

```
uint256[] memory hints = _queue.findCheckpointHints(requestsToClaim, 1, requestsToClaim.length + 1);
```

The hints is fetched using the code above. According to the code and docs:

```
function findCheckpointHints(uint256[] _requestIds, uint256 _firstIndex, uint256 _lastIndex)
    view
    returns (uint256[] hintIds)
```

Requirements:

```
Array of request ids must be sorted
 firstIndex must be greater than 0, because checkpoint list is 1-based array
_lastIndex must be less than or equal to getLastCheckpointIndex()
```

Note how the `_lastIndex` is passed in. The last index is passed as `requestsToClaim.length + 1`, not `getLastCheckpointIndex()`.

- Case 1: `requestsToClaim.length < getLastCheckpointIndex()`:

  Assume user wants to withdraw 5 requestz and the `requestsToClaim.length + 1` will be 6, but the current `getLastCheckpointIndex()` returns 562 (see contract 0x889edc...12f9b1 on Ethereum mainnet).

  Then the hints will always be 0.

- Case 2: `requestsToClaim.length > getLastCheckpointIndex()`:

  This case is rare, but the transaction just reverts when querying the hint. We can work through an example (see transaction 0xcaee15...7f20ba on Ethereum mainnet).

  This is a withdrawal transaction:

  ```
  Request id: 59498
  hints: 555
  ```

  If we query `findCheckpointsHints` in the contract 0x889edc...12f9b1, we pass in:

  ```
  Request id: 59498
  first index: 1
  last index: 2
  ```

  we get 0, the last index is too short and we get no hints. The `getLastCheckpointIndex()` is 562. We pass in:

  ```
  Request id: 59498
  first index: 1
  last index: 562
  ```

  We get the hint value 555 correctly, which matches the withdrawal hint. But if we pass in:

  ```
  Request id: 59498
  first index: 1
  last index: 600
  ```

  In this case, last `index > getLastCheckpointIndex()`, the transaction query just reverts.

Recommendation:

1. Ensure the array of request ids is sorted.
2. Use `getLastCheckpointIndex()` as the last index instead of `requestsToClaim.length + 1`.

Balmy: Given the fact that Lido's request ids are incremental, and how we handle withdrawals, in the adapter, I think it's safe to safe that (1) will always be true, and request ids will be sorted.

Applied recommendation (2) in PR 113.

Cantina Managed: Fix looks good.

3.1.4 Donated reward token can overflow the `yieldAccumulator` and revert user withdraw transaction

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: Let us add this proof of concept to `EarnVault.t.sol`:

```
function test_reward_issue_poc() public {

    uint256 amountToDeposit1 = 1 ether;
    uint256 amountToDeposit2 = 120_000;
```

```
    uint256 amountToDeposit3 = 240_000;
    uint256 amountToDeposit4 = 240_000;
    uint256 amountToReward = 100_000;
    erc20.mint(address(this), 100000 ether);
    uint256[] memory rewards = new uint256[](4);
    uint256[] memory shares = new uint256[](4);
    uint256 totalShares;
    uint256 positionsCreated;
    INFTPermissions.PermissionSet[] memory permissions =
      PermissionUtils.buildPermissionSet(operator, PermissionUtils.permissions(vault.WITHDRAW_PERMISSION()));
    bytes memory misc = "1234";

    address[] memory strategyTokens = new address[](2);
    strategyTokens[0] = address(erc20);
    strategyTokens[1] = address(anotherErc20);
    (StrategyId strategyId, EarnStrategyStateBalanceMock strategy) =
      strategyRegistry.deployStateStrategy(strategyTokens);

    uint256 previousBalance;

    erc20.mint(address(strategy), 1 ether);


    (uint256 positionId1,) =
      vault.createPosition(strategyId, address(erc20), amountToDeposit1, positionOwner, permissions,
      ↪  creationData, misc);
    positionsCreated++;
    anotherErc20.mint(address(strategy), amountToReward);

    anotherErc20.mint(address(strategy), 1 ether);

    (address[] memory tokens, uint256[] memory balances,,) = vault.position(positionId1);

    uint256 amountToWithdraw = balances[0];
    console.log(amountToWithdraw);
    uint256[] memory amounts = new uint256[](2);
    amounts[0] = balances[0];
    amounts[1] = balances[1];

    vm.prank(operator);
    vault.withdraw(positionId1, tokens, amounts, address(this));


  }
```

We run it with

```
forge test -vv --match-test "test_reward_issue_poc"
```

The output is:

```
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 7.07ms (1.88ms CPU time)

Ran 1 test suite in 150.33ms (7.07ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/unit/vault/EarnVault.t.sol:EarnVaultTest
[FAIL: UintOverflowed(1001001001001101101101101101101101101101101101101 [1.001e48],
↪  285449538541191976211657193889899027276549324 7 [2.854e45])] test_reward_issue_poc() (gas: 1908179)
```

Now let us follow the function call:

1. _updateAccountingForRewardToken.
2. _strategyYieldData.update.
3. YieldDataForToken#_update.
4. YieldDataForToken#_encode

```
function _encode(uint256 yieldAccumulator, uint256 balance, uint256 hadLoss) private pure returns
↪  (YieldDataForToken) {
    yieldAccumulator.assertFitsInUint151();
    // slither-disable-next-line unused-return
    balance.toUint104();
    return YieldDataForToken.wrap((yieldAccumulator << 105) | (balance << 1) | hadLoss);
}
```

The yieldAccumulator asserts that the number cannot exceed `uint151`, but after the reward donation, the `yieldAccumulator` exceeds `uint151` and triggers the error (see CustomUintSizeChecks.sol#L14) and reverts the transaction.

```
library CustomUintSizeChecks {
  /// @notice Thrown when a value overflows
  error UintOverflowed(uint256 value, uint256 max);

  uint256 private constant MAX_UINT_151 = 2 ** 151 - 1;

  function assertFitsInUint151(uint256 value) internal pure {
    _verifySize(value, MAX_UINT_151);
  }

  function _verifySize(uint256 value, uint256 max) private pure {
    if (value > max) revert UintOverflowed(value, max);
  }
}
```

When computing the new yield share accumulator (see YieldMath.sol#L88) in the yield math, the yield share accumulator already exceeds `uint151` max value.

Recommendation: Pending.

Balmy: The fix suggested in the finding "Donated token can inflate contract balance and reduce user's deposit worth" also fixes the problem reported here. The fix is implemented in PR 76.

Cantina Managed: Fix looks good. Donated token still inflates spot balance and triggers th zero share deposit error but it is not profitable for an attacker.

3.1.5 `ExternalFee` contract treats newly deposit vault share as yield and charge performance fee on user deposit amount immediately

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: Assume we have a ERC4626 strategy underlying asset is token A, user can choose to:

- Deposit underlying asset, then the code convert the asset to vault share.
- Deposit ERC4626 share directly.

```
  function _connector_deposited(
    address depositToken,
    uint256 depositAmount
  )
    internal
    virtual
    override
    returns (uint256 assetsDeposited)
  {
    IERC4626 vault = ERC4626Vault();
    if (depositToken == _connector_asset()) {
      uint256 shares = vault.deposit(depositAmount, address(this));
      // Note: there might be slippage or a deposit fee, so we will re-calculate the amount of assets deposited
      //       based on the amount of shares minted
      return vault.previewRedeem(shares);
    } else if (depositToken == address(vault)) {
      return vault.previewRedeem(depositAmount);
    } else {
      revert InvalidDepositToken(depositToken);
    }
  }
```

However, the `ExternalFee` contract treats newly deposit vault share (see ExternalFees.sol#L172) as yield and charge performance fee on user deposit immediately.

```
function _fees_deposited(
  address depositToken,
  uint256 depositAmount
)
  internal
  override
  returns (uint256 assetsDeposited)
{
  Fees memory fees = _getFees();
  if (fees.performanceFee == 0) {
    // If performance fee is 0, we will need to clear the last balance. Otherwise, once it's turned on again,
    // we won't be able to understand difference between balance changes and yield
    address asset = _fees_underlying_asset();
    _clearBalanceIfSet(asset);
    return _fees_underlying_deposited(depositToken, depositAmount);
  }

  // Note: we are only updating fees for the asset, since it's the only token whose balance will change
  (address[] memory tokens, uint256[] memory currentBalances) = _fees_underlying_totalBalances();
  uint256 performanceFees = _calculateFees(tokens[0], currentBalances[0], fees.performanceFee);

  assetsDeposited = _fees_underlying_deposited(depositToken, depositAmount);

  _performanceData[tokens[0]] = PerformanceData({
    // Note: there might be a small wei difference here, but we can ignore it since it should be negligible
    lastBalance: (currentBalances[0] + assetsDeposited).toUint128(),
    performanceFees: performanceFees.toUint120(),
    isSet: true
  });
}
```

Assume there is no asset in the stragety, when depositing the underlying token, the `_fees_underlying_totalBalances` returned `currentBalances[0]` is 0

Assume there is no asset in the strategy, when depositing the vault share, the `_fees_underlying_totalBalances` returned `currentBalances[0]` is the user deposit share amount.

The proof of concept is located in LidoStrategyEarnVault.t.sol#L392.

Note, we modify the performance fee to 500 bps

```
vm.mockCall(
  address(feeManager), abi.encodeWithSelector(IFeeManagerCore.getFees.selector), abi.encode(Fees(0, 0, 500, 0))
);
```

Run the proof of concept with:

```
forge test -vv --match-path test/unit/vault/LidoStrategyEarnVault.t.sol --match-test
↪  "test_deposit_share_charged_performance_fee" --fork-url [rpc-url]
```

we see that the user's deposit fund is immediately reduced.

The initial deposit amount is:

```
uint256 amountToDeposit1 = 100_000;
uint256 amountToDeposit2 = 200_000;
uint256 amountToDeposit3 = 50_000;
uint256 amountToReward = 100_000;
```

After the stETH, the user deposit becomes:

```
position user 1 93389
position user 2 196609
position user 3 49999
position user 4 49999
```

Recommendation: Before the deposit / withdraw / special withdraw to not count new vault share as yield.

Balmy: During a deposit, we are currently sending the user's tokens directly to the strategy, and then we'll let it know about the transfer. We do this as an optimization as to have only one transfer (user ⇒ strategy) instead of two (user => vault => strategy). The thing is that this approach doesn't let the strategy "react" to the deposit. They can't check balances before the deposit is made, causing the problem described here. So we are going to change the approach and use two transfers.

First part of the change is in PR 77 and the second part of the change is in PR 133.

3.2 Medium Risk

3.2.1 Malicious actor can frontrun the strategy registration to take over strategy ownership

Severity: Medium Risk

Context: EarnStrategyRegistry.sol#L43-L54

Description: 1. Alice create a strategy and wants to register strategy and assign alice address as the first owner.

2. Bob frontrun the transaction and set bob as the first owner.
3. Bob propose a strategy migration.
4. After 3 days migration timelock, Bob executes the migration and transfer the token in the stragety out to a new malicious contract.

Recommendation: Return the first owner from the strategy contract to avoid such frontrunning.

```
    owner[strategyId] = strategy.firstOwner(); // <-- fix here
     _nextStrategyId = strategyId.increment();
     emit StrategyRegistered(firstOwner, strategyId, strategy);
     strategy.strategyRegistered(strategyId, IEarnStrategy(address(0)), "");
```

Balmy: We decided to move the responsibility of registration to the strategies. So the idea would be to basically change `registerStrategy` to the following:

```
function registerStrategy(address firstOwner) external returns (StrategyId strategyId) {
  IEarnStrategy strategy = IEarnStrategy(msg.sender);
  _revertIfNotStrategy(strategy);
  // ...
}
```

By doing this, we are only allowing strategies to register themselves, so frontunning isn't possible. The strategy builders would need to write code specially for registration, but we think it's a little more flexible in the sense that:

- It wouldn't be a function that remains public for the rest of the strategy's life.
- It can easily be added to the constructor/initialization function.
- Or it could be something else that the strategy wants to implement, like a `register` function on the strategy, that implements access checks.

The fix in core is in PR 70, and the fix in periphery is in PR 105.

Cantina Managed: Fixed.

3.2.2 `CompoundV2Connector#_convertSharesToAssets` does not consider pending `cToken` interest

Severity: Medium Risk

Context: CompoundV2Connector.sol#L378-L383

Description: The code attempts to convert the share balance to asset worth ⇒ convert cToken balance to asset worth.

```
uint256 magnitude = (10 + underlyingTokenDecimals);
return shares.mulDiv(cToken().exchangeRateStored(), 10 ** magnitude, rounding);
```

However, the code use `exchangeRateStored()` instead of `exchangeRateCurrent()` (see CToken.sol#L274):

```
/**
  * @notice Accrue interest then return the up-to-date exchange rate
  * @return Calculated exchange rate scaled by 1e18
  */
function exchangeRateCurrent() override public nonReentrant returns (uint) {
    accrueInterest();
    return exchangeRateStored();
}

/**
  * @notice Calculates the exchange rate from the underlying to the CToken
  * @dev This function does not accrue interest before calculating the exchange rate
  * @return Calculated exchange rate scaled by 1e18
  */
function exchangeRateStored() override public view returns (uint) {
    return exchangeRateStoredInternal();
}
```

The `exchangeRateStored` does not consider the pending interest and considered as a lagging interest rate.

Proof of Concept: The proof of concept below demonstrates the differences. Note that we use the formula below to compute the `cToken` value:

```
cToken balance * exchangeRate current / (10 ** token decimals)
```

vs

```
cToken balance * exchangeRate Stored / (10 ** token decimals)
```

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

interface ICERC20 is IERC20 {
    function mint(uint256 underlyingAmount) external returns (uint256);
import "forge-std/console.sol";
    function redeemUnderlying(uint256 underlyingAmount) external returns (uint256);
    function exchangeRateStored() external view returns (uint256);
    function decimals() external view returns (uint256);
    function getCash() external view returns (uint256);
    function totalReserves() external view returns (uint256);
    function totalBorrows() external view returns (uint256);
    function redeem(uint redeemTokens) external returns (uint);
    function exchangeRateCurrent() external returns (uint);
  }

  interface IComptroller {
    function claimComp(address[] memory holders, ICERC20[] memory cTokens, bool borrowers, bool suppliers)
    ↪  external;
    function compSpeeds(address cToken) external view returns (uint256);
    function compAccrued(address) external view returns (uint256);
    function mintGuardianPaused(ICERC20 cToken) external view returns (bool);
  }


contract CounterTest is Test {

    ICERC20 cToken = ICERC20(0x5d3a536E4D6DbD6114cc1Ead35777bAB948E3643); // cDAI
    IERC20 asset = IERC20(0x6B175474E89094C44Da98b954EedeAC495271d0F); // DAI
    IERC20 comp = IERC20(0xc00e94Cb662C3520282E6f5717214004A7f26888); // COMP

    address cTokenHolder = 0xB0b0F6F13A5158eB67724282F586a552E75b5728; // cDAI holder

    using stdStorage for StdStorage;
    StdStorage stdlib;

    function setUp() public {

    }

    function testRedeem() public {
```

```
        address user = 0x90581aFC83520a649376852166B3df92153cEE20;

        vm.startPrank(user);

        uint256 balance = cToken.balanceOf(user);

        console.log("balance: %s", balance);

        uint256 daiBefore = asset.balanceOf(user);

        uint256 expected_redeem_dai = balance * cToken.exchangeRateStored() / 10 ** 18;

        console.log("redeem dai with lagging interest: %s", expected_redeem_dai);

        uint256 accurate_redeem_dai = balance * cToken.exchangeRateCurrent() / 10 ** 18;

        console.log("accurate redeem dai             : %s", accurate_redeem_dai);

        cToken.redeem(balance);

        uint256 daiAfter = asset.balanceOf(user);

        console.log("redeemed dai balance change     : %s", daiAfter - daiBefore);

    }
}
```

Run the test wit the following command:

```
forge test -vvv --match-test "testRedeem" --fork-url "https://eth.llamarpc.com" --block-number 21182834
```

The output is:

```
Ran 1 test for test/Counter.t.sol:CounterTest
[PASS] testRedeem() (gas: 194959)
Logs:
  balance: 12014076301212601
  redeem dai with lagging interest: 28658075989529341095539 94
  accurate redeem dai             : 28658306184303684899284 06
  redeemed dai balance change     : 28658306184303684899284 06
```

As we can see, the real redeemed DAI is 286583061843036848992840 6. Yet without consider the pending interest, the computed DAI balance from cToken is 286580759895293410955399 4.

Recommendation: Use `exchangeRateCurent()`, or if the protocol wants to keep the view function, consider calling `accureInterest` (see CToken.sol#L327) first become use the exchange rate data.

Balmy: You are right that we are not considering pending interest. We looked into existing solutions, and we think that using transmissions11's LibCompound would be a good solution (we made a small change to use OpenZeppelin's math library instead of Solmate's one). We need to keep the function `view`, because we need to use it for `totalBalances()`.

We've also seen this library being used in Yield Daddy's ERC4626 adapter (audited by yaudit), and Moonwell's ERC4626 adapter (audited by Halborn).

At the same time, this change should also fix `CompoundV2Connector#_convertSharesToAssets` magnitude scalar can be too large and truncate the asset worth.

Fix is in PR 101.

Cantina Managed: Fixed.

3.2.3 Protocol migration does not transfer the underlying asset to the new strategy contract when the rescue is completed

Severity: Medium Risk

Context: ExternalGuardian.sol#L83-L87

Description: When rescue starts, the _guardian_underlying_withdraw (see ExternalGuardian.sol#L84) withdraws underlying funds back to the strategy contract.

```
(tokens, rescued) = _guardian_underlying_maxWithdraw();
IEarnStrategy.WithdrawalType[] memory types = _guardian_underlying_withdraw(0, tokens, rescued, address(this));
if (!_areAllImmediate(types)) {
  revert OnlyImmediateWithdrawalsSupported();
}
```

For example, if the strategy is AAVE V3, xxD:

- User deposits underlying asset in exchange for aToken.
- When withdrawing, aToken in the contract is burnt and underlying asset is transferred back.

Yet when the strategy s migrated, only the aToken is transferred (see AaveV3Connector.sol#L319):

```
function _connector_migrateToNewStrategy(
  IEarnStrategy newStrategy,
  bytes calldata
)
  internal
  override
  returns (bytes memory)
{
  IERC20 vault_ = aToken();
  uint256 balance = vault_.balanceOf(address(this));
  vault_.safeTransfer(address(newStrategy), balance);
  return abi.encode(balance);
}
```

The compound V2 strategy follows the same pattern, only `cToken` is transferred out.

However, consider the following sequence of actions:

1. The strategy owner propose a strategy migration.
2. The migration is subject to a 3 days time-lock (see EarnStrategyRegistry.sol#L110).
3. During these 3 days, a rescue transaction is executed, all aToken is burnt to withdraw underlying token.
4. Migration is executed, yet no underlying asset is transferred to the new strategy.

Recommendation: While executing the rescue in the 3 days timelock is an edge case, the protocol can consider:

- Transfer both aToken and underlying token to the new strategy contract in AAVE V3 connector.
- Transfer both cToken and underlying token to the new strategy contract in Compound V2 connector.
- Transfer both vault share and vault underlying asset to the new strategy contract in ERC4626 connector.

Balmy: We've decided to simply disable migrations on strategies that:

- Have an ongoing rescue (one that still needs confirmation).
- Have a confirmed rescue.

The fix is in PR 109.

Cantina Managed: Fixed.

3.2.4 Connectors special withdraw rounding amount direction should not favor user

Severity: Medium Risk

Context: CompoundV2Connector.sol#L295-L334

Description: When a user withdraws, the rounding direction should always favor the protocol and not the user to make sure that user cannot take advantage of such rounding error.

Yet Both compound V2 special withdraw and ERC4626 special withdraw favor the user when computing cTokens and shares.

In Compound V2 Connector:

```
uint256 shares = _convertAssetsToShares(assets, Math.Rounding.Ceil);
```

The rounding direction is Ceil instead of floor. In ERC4626 connector special withdraw when the type is `WITHDRAW_ASSET_FARM_TOKEN_BY_ASSET_AMOUNT`.

The `previewWithdraw` (see ERC4626Connector.sol#L217) from a standard vault (see ERC4626.sol#L161) round up as well.

```
/** @dev See {IERC4626-previewWithdraw}. */
function previewWithdraw(uint256 assets) public view virtual returns (uint256) {
    return _convertToShares(assets, Math.Rounding.Ceil);
}
```

Recommendation:

1. For Compound V2, use `Math.Rounding.Floor` as rounding direction.
2. For ERC6462 connector, use `vault.convertToShares(assets)` instead of `previewWithdraw`.

Balmy: The fix for the Compound connector was actually in the fix for `CompoundV2Connector#_convertSharesToAssets` does not consider pending `cToken` interest (PR 101).

You are also right that we should not use `previewWithdraw` for the ERC4626. But not only for rounding, we should not be using it because the vault could have fees enabled. I tested out a quick example based on this implementation (ERC4626Fees.sol):

```
ERC4626 vault
total shares in vault = 2000
total assets in vault = 4000
withdraw fee = 5%

Strategy
total shares owned by strategy = 500
reported balance = previewRedeem(500) = 500 * 4000 / 2000 * 0.95 = 950

Vault
total shares for strategy = 1000
positon shares = 500
position balance = 950 * 500 / 1000 = 475


Assuming we are using previewWithdraw:
previewWithdraw(200) = (200  * 1.05) * 2000 / 4000 = 105
redeem(105) = (105 * 4000 / 2000) * 0.95 = 199.5

Assuming we are using convertToShares:
convertToShares(200) = 200 * 2000 / 4000 = 100
redeem(100) = (100 * 4000 / 2000) * 0.95 = 190
```

So indeed, `convertToShares` is the way to go. The fix is in PR 114.

Cantina Managed: Fixed.

3.2.5 Campaign creation reverts if exact native token amount not sent

Severity: Medium Risk

Context: LiquidityMiningManager.sol#L153-L161

Description: `setCampaign()` requires depositing reward token that will be disbursed to LPs. It depends on the current balance of the campaign which in turn depends on the current time (`block.timestamp`). Thus, depending on when this transaction is executed, the amount of native token to send with the call changes.

`setCampaign()` reverts if the exact amount of native token calculated isn't set which, as explained above, depends on the time of executing the transaction. This makes it difficult to estimate leading to high chances of a revert.

Recommendation: Instead of reverting if `msg.value` isn't exactly as calculated, continue the execution and refund the extra amount to the caller.

Balmy: Fixed in PR 120.

Cantina Managed: Fixed.

3.2.6 Consider add reentrancy protection to the `specialWithdrawFees` function

Severity: Medium Risk

Context: ExternalFees.sol#L66-L92

Description: The function withdraw fee calls `_updateFeesForWithdraw` before executing the token transfer, However, the function `specialWithdrawFees` updates the fee state after making the external call.

If the special withdrawal token (a ERC777 token) has a callback function, a malicious caller can re-enter the `specialWithdrawFees` function.

Recommendation: Follow checks-effects-interactions pattern and update the state before the external transfer.

```
    _updateFeesForWithdraw({
    tokens: tokens,
    withdrawAmounts: balanceChanges,
    currentBalances: currentBalances,
    fees: fees
  });

(balanceChanges, actualWithdrawnTokens, actualWithdrawnAmounts, result) =
    _fees_underlying_specialWithdraw(0, withdrawalCode, toWithdraw, withdrawData, recipient);
```

Balmy: We can't follow check effect interaction per your suggestion since `_updateFeesForWithdraw` uses values returned by `_fees_underlying_specialWithdraw`. So they would have to be executed in that order.

However, I think it makes sense to add a reentrancy lock. Done here in PR 124.

Cantina Managed: Fixed.

3.2.7 Protocol reported: liquidity mining layer doesn't increase the length of the `balanceChanges` array during special Withdrawal

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: During a special withdrawal, the liquidity mining layer doesn't increase the length of the `balanceChanges` array properly.

Let's see an example:

- The connector supports only token USDC.
- The liquidity mining layer adds OP as a reward token.
- The liquidity mining layer added OP in almost all functions:
  - `allTokens`.
  - `withdraw`.

But we aren't modifying the length in `balanceChanges` when `specialWithdraw` is called.

In our example, it would have length 1, while `allTokens` would have length 2, which breaks the earn vault accounting.

Balmy: Fixed in PR 100.

Cantina Managed: Fixed.

3.2.8 User can pass in zero underlying token amount and non-zero reward amount to bypass the fee accumulation

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: If the strategy has token A as base underlying asset, and token B as reward token, the protocol wants to charge performance fee both on the yield part of underlying asset and the reward token balance.

But if a user passes in withdraw amount `[0, 100]` the withdrawal fee (see ExternalFees.sol#L207) for reward token will be skipped because when the first token (underlying token) is 0, no fee will be charged nor accumulated.

```
(, uint256[] memory currentBalances) = _fees_underlying_totalBalances();
for (uint256 i; i < tokens.length; ++i) {
  // If there is nothing being withdrawn, we can skip fee update, since balance didn't change
  if (toWithdraw[0] == 0) continue;
```

Recommendation:

```
if (toWithdraw[i] == 0) continue;
```

Balmy: Fixed in PR 131.

Cantina Managed: Fix looks good.

3.2.9 Potential read-only entrancy pattern if external protocol read the share state from the `EarnVault`

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: In `Vault#withdraw` function, the code updates the state after an external call (see EarnVault.sol#L293):

```
withdrawalTypes = strategy.withdraw({
  positionId: positionId,
  tokens: tokensToWithdraw,
  toWithdraw: withdrawn,
  recipient: recipient
});

// slither-disable-next-line unused-return
(, uint256[] memory balancesAfterUpdate) = strategy.totalBalances();

_updateAccounting({
```

The withdraw function and all functions in Vault are certainly guarded by a `nonReentrant` modifier, such pattern is set up for read-only reentrancy.

- Sentiment's hackmd shows an example of such hack caused by balancer read-only reentrancy.
- Quillaudits also shows an example of such hack caused by curve read-only reentrancy.

Recommendation: At least if there is any external integration that read share value from the earn vault, they should be aware there is this read-only reentrancy vector.

Balancer adds a read-only reentrancy check (see VaultReentrancyLib.sol#L19) when calling the view related function, a similar pattern can be followed.

Balmy: Fixed in PR 78.

Cantina Managed: Fix looks good.

3.3 Low Risk

3.3.1 `LidoSTETHConnector#_connector_deposit` returns ETH amount instead of stETH received amount

Severity: Low Risk

Context: LidoSTETHConnector.sol#L63-L77

Description: The strategy (and connector) are meant to return the amount of assets deposited. If a user deposits ETH, the ETH is submitted for stETH.

Yet the ETH amount is returned, this returned amount is used to compute the deposit fee, etc...

The total balance however, returns the stETH balance (see LidoSTETHConnector.sol#L139).

```
function _connector_totalBalances()
  internal
  view
  override
  returns (address[] memory tokens, uint256[] memory balances)
{
  tokens = _connector_allTokens();
  balances = new uint256[](tokens.length);
  balances[0] = IERC20(address(_stETH)).balanceOf(address(this));
}
```

However, if stETH depegs from ETH severely, the return deposit amount will not be accuate (see a related post by huobi research).

In the past, the stETH depegged nearly 5% from ETH. Even when the stETH does not depegg too much from ETH, the internal lido conversion rate from ETH to stETH is not 1:1.

Recommendation:

```
if (depositToken == _connector_asset()) {
  // slither-disable-next-line unused-return
  uint256 balanceBefore = stETH.balance(address(this));
  _stETH.submit{ value: depositAmount }(address(0));
  uint256 balanceAfter = stETH.balance(address(this));
  return balanceAfter - balanceBefore;
}
```

Balmy: Fixed in PR 102.

Cantina Managed: Fixed. You can also just use the return value of `_stETH.submit`. It returns the Amount of stETH shares generated.

3.3.2 `estimatedPendingFunds` and `withdrawableFunds` in Lido delayed withdrawal adapter should return 0 if token is not ETH

Severity: Low Risk

Context: LidoSTETHDelayedWithdrawalAdapter.sol#L73-L99

Description: In

```
function estimatedPendingFunds(uint256 positionId, address)
function withdrawableFunds(uint256 positionId, address)
```

the first parameter is the position id, the second parameter (DelayedWithdrawalManager.sol#L44) is the token address and it is not used.

When calling `estimatedPendingFunds` and `withdrawableFunds`, the code returns the amount of pending stETH regardless of what the token query is.

```
function estimatedPendingFunds(uint256 positionId, address token) public view returns (uint256 pendingFunds) {
  mapping(uint256 index => RegisteredAdapter registeredAdapter) storage registeredAdapters =
    _registeredAdapters.get(positionId, token);
  uint256 i = 0;

  bool shouldContinue = true;
  while (shouldContinue) {
    RegisteredAdapter memory adapter = registeredAdapters[i];
    if (address(adapter.adapter) != address(0)) {
      // slither-disable-next-line calls-loop
      pendingFunds += adapter.adapter.estimatedPendingFunds(positionId, token); // <==
      unchecked {
        ++i;
      }
    }
    shouldContinue = adapter.isNextFilled;
  }
}
```

Recommendation: Check if the token is ETH, if the token is not ETH, return 0.

Balmy: Fixed in PR 111. We also took the opportunity to add reverts if the token wasn't ETH during `initiateDelayedWithdrawal` and `withdraw`.

Cantina Managed: Fixed.

3.3.3 Reward generation can be sandwiched for unfair allocation to a position

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: For strategies that increase their rewards in discrete time steps, the yield generation event can be sandwiched to get more allocation in that yield.

Say 1000 reward tokens are going to be allocated to the strategy, an attacker can:

- Frontrun this transaction to create a position and increase their number of shares.
- The strategy gets the rewards. At this time, both honest users and attacker have shares so the reward gets allocated according these shares.
- The attacker withdraws their allocation of the reward token.

Since the rewards were generated based on the deposit of honest users, the attacker shouldn't get a portion of these rewards.

`AaveV3Connector.claimAndDepositAssetRewards()` is an example where the yield in asset reward will jump and this can be sandwiched if this is not handled in aaveV3 strategy.

Proof of Concept: Add the following in `EarnVault.t.sol`:

```
import "forge-std/Test.sol";

contract EarnVaultTest is PRBTest, StdCheats, StdUtils {

address alice;

function setUp() public {
   ...
  alice = makeAddr("alice");
  erc20.mint(alice, 1000e18);
  vm.label(alice, "alice");
}

function test_sandwichRewards() public {
  uint256 amountToDeposit1 = 100_000;
  uint256 amountToBurn = 1000;
  uint256 amountToReward = amountToBurn;
  erc20.mint(address(this), 1000e18);
  INFTPermissions.PermissionSet[] memory permissions =
    PermissionUtils.buildPermissionSet(operator, PermissionUtils.permissions(vault.WITHDRAW_PERMISSION()));
  bytes memory misc = "1234";

  address[] memory strategyTokens = new address[](2);
  strategyTokens[0] = address(erc20);
  strategyTokens[1] = address(anotherErc20);
  (StrategyId strategyId, EarnStrategyStateBalanceMock strategy) =
    strategyRegistry.deployStateStrategy(strategyTokens);

  // honest user creates position
  (uint256 positionId1,) =
    vault.createPosition(strategyId, address(erc20), amountToDeposit1, positionOwner, permissions,
    ↪  creationData, misc);

  // rewards earned
  anotherErc20.mint(address(strategy), amountToReward);

  vm.warp(block.timestamp + 24 hours);

  vm.startPrank(alice);
  erc20.approve(address(vault), type(uint256).max);

  // alice (attacker) sandwiches the next reward generation event by creating a position and then withdrawing
```

```
  (uint256 positionId2,) =
    vault.createPosition(strategyId, address(erc20), amountToDeposit1, alice, permissions, creationData, misc);
  vm.stopPrank();

  // reward earned
  anotherErc20.mint(address(strategy), amountToReward);

  (address[] memory tokens1, uint[] memory balances1,,) = vault.position(positionId1);
  (, uint[] memory balances2,,) = vault.position(positionId2);


  assertEq(balances1[1], 1500); // position1 should get all the rewards (2000)
  assertEq(balances2[1], 500); // position2 should get no rewards (0)

  vm.startPrank(alice);
  // sandwich complete
  vault.withdraw(positionId2, tokens1, balances2, alice);
  vm.stopPrank();
}
```

Recommendation: Document that strategies shouldn't have discrete yield generation events. If they do, they should allocate the yield to the vault in a continuously streaming fashion, so that a large yield generation event cannot be sandwiched.

Balmy: I don't think this should be a responsibility of vault. I think it should be up to each strategy to implement a continuous allocation of funds. Specially since the vault would need the strategy to report extra information about such yield generation event (like when it happened, or when the next one will happen). If the strategy had this information, then it could simply convert it to a continuous allocation.

Yes, `AaveV3Connector` is actually a case where we could get sandwiched, but I think we can live with it. We haven't seen a case where the Aave v3 provides rewards in the same token as the one being deposited. We added this because their code doesn't actually prevent it from happening, but we haven't seen it "in real life" yet and is unlikely to happen.

If it does, I think we'd rather mitigate it by calling this function often so that the sandwich is less attractive, rather than figuring out a way to distribute it over time.

We added comments in PR 75.

Cantina Managed: Acknowledged.

3.3.4 Donated token can inflate contract balance and reduce user's deposit worth

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In all strategy contracts, the `_connector_totalBalances` queries the spot balance (see ERC4626Connector.sol#L81) using `balanceOf`.

```
function _connector_totalBalances()
  internal
  view
  virtual
  override
  returns (address[] memory tokens, uint256[] memory balances)
{
  IERC4626 vault = ERC4626Vault();
  tokens = new address[](1);
  tokens[0] = _connector_asset();
  balances = new uint256[](1);
  balances[0] = vault.previewRedeem(vault.balanceOf(address(this)));
}
```

However, users can inflate the total balance in the contract and thus reduce user's deposit share and user asset worth.

Proof of Concept:

```
function test_donation_issue_poc() public {

  uint256 amountToDeposit1 = 1 ether;
```

```
  uint256 amountToDeposit2 = 120_000;
  uint256 amountToDeposit3 = 240_000;
  uint256 amountToDeposit4 = 240_000;
  uint256 amountToReward = 120_000;
  erc20.mint(address(this), 100000 ether);
  uint256[] memory rewards = new uint256[](4);
  uint256[] memory shares = new uint256[](4);
  uint256 totalShares;
  uint256 positionsCreated;
  INFTPermissions.PermissionSet[] memory permissions =
    PermissionUtils.buildPermissionSet(operator, PermissionUtils.permissions(vault.WITHDRAW_PERMISSION()));
  bytes memory misc = "1234";

  address[] memory strategyTokens = new address[](2);
  strategyTokens[0] = address(erc20);
  strategyTokens[1] = address(anotherErc20);
  (StrategyId strategyId, EarnStrategyStateBalanceMock strategy) =
    strategyRegistry.deployStateStrategy(strategyTokens);

  uint256 previousBalance;

  erc20.mint(address(strategy), 200 ether);

  (uint256 positionId1,) =
    vault.createPosition(strategyId, address(erc20), amountToDeposit1, positionOwner, permissions,
    ↪  creationData, misc);
  positionsCreated++;
  anotherErc20.mint(address(strategy), amountToReward);

  (address[] memory tokens, uint256[] memory balances,,) = vault.position(positionId1);

  uint256 amountToWithdraw = balances[0];
  uint256[] memory amounts = new uint256[](2);
  amounts[0] = balances[0];
  amounts[1] = balances[1];

  // loop over and log balances

  for (uint256 i; i < balances.length; i++) {
    console.log(balances[i]);
  }

  uint256 balanceBefore = IERC20(tokens[0]).balanceOf(address(this));

  vm.prank(operator);
  vault.withdraw(positionId1, tokens, amounts, address(this));

  console.log(1 ether);
  (tokens, balances,,) = vault.position(positionId1);

  for (uint256 i; i < balances.length; i++) {
    console.log(balances[i]);
  }

  assertEq(IERC20(tokens[0]).balanceOf(address(this)), balanceBefore + amountToWithdraw);

}
```

Add this test to `EarnVault.t.sol` and run it with:

```
forge test -vv --match-test "test_donation_issue_poc"
```

Basically a user frontruns the create Position and donates 200 tokens to the strategy contract directly.

```
erc20.mint(address(strategy), 200 ether);
```

User mints shares using 1 ETH, yet they only get 0.8 ETH back, meaning that after a donation, users lose 20% of their funds.

```
One ETH.         : 1000000000000000000
User withdraw:  800796812749003984
```

Recommendation: Track the deposit using an internal variable instead of spot balance to avoid such donation issue. While such attack may not be profitable for the attacker, the inflation can be considered

as a griefing vector because the user's asset worth is reduced.

Balmy: The recommendation said to use an internal variable instead of sport balance when calculating the balance in our strategies. While possible:

- It would be a big change, since we would need to track deposits, withdrawals, special withdrawals, migrations.
- It's difficult to do with strategies like Aave's, where their tokens are rebasing.

We would like to suggest another change. Since the attack only seems feasible when the strategy is empty, we could handle by ignoring all assets when the strategy is empty. So, if total shares is 0, all assets must have been donated, so we could handle it like this:

```
function convertToShares(
  uint256 assets,
  uint256 totalAssets,
  uint256 totalShares,
  Math.Rounding rounding
)
  internal
  pure
  returns (uint256)
{
  if (totalShares == 0) {
    totalAssets = 0;
  }
  return assets.mulDiv(totalShares + SHARES_OFFSET_MAGNITUDE, totalAssets + 1, rounding);
}
```

With this change, the attacker can deposit `1e28` tokens (`1e10` times more than the user) and there would be no balance loss. The first depositor would actually get the donation as balance, so they would be quite happy. It has been implemented in PR 76.

Cantina Managed: Ww tested the solution, the fix is ok. A donation no longer causes issues for reward accounting and share accounting.

3.3.5 Compound V2 strategy has no reward generated

Severity: Low Risk

Context: CompoundV2Connector.sol#L149-L159

Description: While the reward balance is computed below:

```
balances[1] = comp().balanceOf(address(this)) + comptroller().compAccrued(address(this));
```

The compound v2 cToken ceases to accure any compound reward. The compound token reward is distributed to the Compound V3 only.

Recommendation: Consider integrating with compound v3 in the future to accure COMP token reward.

Balmy: While true, there are some Compound forks that implement the same interface and so provide their tokens as rewards. So we would like to keep that part of the code.

Cantina Managed: Acknowledged.

3.3.6 Protocol reported: Companion contract can overwrite the validation data

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: While the current code worked, it would only work with the current ToS validation.

If the code cannot perform any other validation in the future because the Companion would overwrite the data (see EarnVaultCompanion.sol#L83).

Recommendation: The code should not overwrite the creation validation data anymore.

Balmy: Fixed in PR 128.

Cantina Managed: Fixed.

3.3.7 Consider adding reentrancy protection for `DelayedWithdrawalManager#withdraw`

Severity: Low Risk

Context: DelayedWithdrawalManager.sol#L122-L128

Description: The function updates the state after making the external call `withdraw`, if the receiver can re-enter the function when calling `adapter.adapter.withdraw`, the code is vulnerable to reentrancy:

```
do {
  RegisteredAdapter memory adapter = registeredAdapters[i];
  if (address(adapter.adapter) != address(0)) {
    // slither-disable-next-line calls-loop
    (uint256 _withdrawn, uint256 _stillPending) = adapter.adapter.withdraw(positionId, token, recipient); //
    ↪  <-- external call
    withdrawn += _withdrawn;
    stillPending += _stillPending;
    if (_stillPending != 0) {
      if (i != j) {
        registeredAdapters.set(j, adapter.adapter);
      }
      unchecked {
        ++j;
      }
    }
    unchecked {
      ++i;
    }
  }
  shouldContinue = adapter.isNextFilled;
} while (shouldContinue);
registeredAdapters.pop({ start: j, end: i }); // <--- state update.
```

Recommendation: Consider adding a reentrancy guard to `DelayedWithdrawalManager#withdraw`.

Balmy: Fixed in PR 132. I'm also adding a reentrancy guard to `registerDelayedWithdraw`, since the caller could register themselves again and mess up the `.pop` somehow.

Cantina Managed: Fixed.

3.3.8 Cloned LIDO `LidoSTETHStrategy.sol` is not capable of receiving ETH

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: If the Strategy is `LidoSTETHStrategy.sol`, a user can choose to

1. Deposit ETH, the code submit ETH in exchange for stETH.
2. Deposit stETH directly.

However, if the user chooses to deposit ETH, the ETH is sent to `EarnVault` and the `EarnVault` contract sends ETH (see EarnVault.sol#L580) to the Cloned `LidoSTETHStrategy` contract:

```
depositToken.transferIfNativeOrTransferFromIfERC20({ recipient: address(strategy), amount: depositedAmount });
assetsDeposited = strategy.deposited(depositToken, depositedAmount);
```

However, the cloned strategy connot receive the ETH.

Proof of Concept: Run the proof of concept in LidoStrategyEarnVault.t.sol#L360 with the command:

```
forge test -vvvv --match-path test/unit/vault/LidoStrategyEarnVault.t.sol --match-test
↪  "test_cloned_strategy_cannot_receive_ETH" --fork-url [eth rpc url]
```

The output is:

```
            ← [Return] Fees({ depositFee: 0, withdrawFee: 0, performanceFee: 0, rescueFee: 0 })
          [0]
↪  PRECOMPILES::ecmul(23987658082447202574735589866264198368024857023092404036088857803237646925824,
↪  86271829334882047342934448278462818155638862152129831939531552797 4912,
↪  22646313910175617102577115759972643379895361804348396664382073864 1920)
            ← [Return]
          emit Initialized(version: 1)
          ← [Stop]
        ← [Return]
    ← [Return] 0x45C92C2Cd0dF7B2d705EF12CfF77Cb0Bc557Ed22
 [982] 0x45C92C2Cd0dF7B2d705EF12CfF77Cb0Bc557Ed22::fallback{value: 1000000000000000000}()
    [801] LidoSTETHStrategy::c7183455{value: 1000000000000000000}(a4c133ae270771860664b6b7ec320bb1000000000000
↪  0000000000000000000000000415cf58144ef33af1e14b5208015d11f9143e27b9003c) [delegatecall]
      ← [Revert] EvmError: Revert
    ← [Revert] EvmError: Revert
 emit LogNamedString(key: "Error", value: "Lido Strateg failed to receive ETH")
 emit Log(err: "Error: a == b not satisfied [bool]")
```

Recommendation: If the cloned strategy cannot receive ETH, the user can always to choose swap ETH for stETH in companion contract and then deposit stETH. The protocol can consider only support stETH deposit in `LidoSTETHStrategy.sol`.

Balmy: After some digging, the problem seems to be the library we were using to deploy clones: `wighawag/clones-with-immutable-args`. So we migrated away from that library, and we started using OpenZeppelin's library instead. By doing that, now our clones are able to receive ETH without reverting. You can see the fix in PR 134.

Cantina Managed: Fix looks good, we recommend adding more testing to check if all the strategy contracts are fetched correctly using the clone library.

3.3.9 User loses funds if they deposit to a malicious strategy

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: If the user chooses to deposit fund to a strategy, the token is sent to `EarnVault` and the `EarnVault` contract sends tokens (EarnVault.sol#L580) to the strategy.

```
depositToken.transferIfNativeOrTransferFromIfERC20({ recipient: address(strategy), amount: depositedAmount });
assetsDeposited = strategy.deposited(depositToken, depositedAmount);
```

However, if the strategy is a malicious contract, the malicious contract can transfer the user's funds out instead of providing yield.

Recommendation: Just like uniswap, a malicious actor deploys a malicious token pool, then the malicious actor can deploy malicious strategy. The protocol should warn users that they need to choose the strategy contract carefully.

Balmy: Yes, this is true. Our plan is to only show a curated list of strategies on our UI. We already explain the following on the README:

> In Earn, a user deposits an "asset" and immediately starts generating yield in one or more tokens (one of these tokens could also be the same asset they deposited). When a user deposits their funds, they can choose the "strategy" they'd like to use to generate this yield. Earn strategies are in charge of taking the asset and start generating yield, so they will be the ones having control over all user funds. It will be up to each user to do their own due diligence and select their preferred strategy, based on their own risk/reward inclinations.

Cantina Managed: Yes, both the readme highlights and selective UI display are a good approach.

3.3.10 Manager gets different views of strategy's balance

Severity: Low Risk

Context: ExternalLiquidityMining.sol#L134-L162

Description: These functions call `manager.withdrew()` and withdraw underlying token in the opposite order. It would be good to be consistent here. The manager may depend on strategy existing balance.

Recommendation: Make this order consistent by first withdrawing tokens and then calling the manager.

Balmy: Fixed in PR 121.

Cantina Managed: Fixed.

3.4 Gas Optimization

3.4.1 Use `do {...} while` loop instead of a simple `while` loop

Severity: Gas Optimization

Context: DelayedWithdrawalManager.sol#L138

Description: `while (shouldContinue) {...}` can be converted to a `do {...} while (shouldContinue);` since the first iteration always happens. Check the solidity docs for reference.

Recommendation: Use a do-while loop instead of a while loop.

Balmy: Fixed in PR 127.

Cantina Managed: Fixed.

3.4.2 `rewardBalance` is set but not used

Severity: Gas Optimization

Context: CompoundV2Connector.sol#L286

Description: Here, `rewardBalance` is set but isn't used later.

Recommendation: Skip setting `rewardBalance`.

Balmy: We already removed it as part of another fix in PR 101.

Cantina Managed: Verified.

3.4.3 No need to compute `newBalance`

Severity: Gas Optimization

Context: EarnVault.sol#L749-L757

Description: `newBalance` is set to `calculatedData.positionBalance - withdrawn` which is already computed and saved in `newPositionBalance`. You can save gas by assigning it to `newBalance` directly instead of recomputing the value.

Recommendation: Apply this diff:

```
- newBalance: calculatedData.positionBalance - withdrawn,
+ newBalance: newPositionBalance,
```

Balmy: Fixed in PR 77.

Cantina Managed: Verified.

3.4.4 Quadratic order complexity loops for tokens

Severity: Gas Optimization

Context: EarnStrategyRegistry.sol#L166-L171, Utils.sol#L14-L27

Description: Strategy contracts store tokens (asset and yield tokens) in an array. When migrating to a new strategy, it is checked that th e new strategy's tokens are a superset of the current strategy tokens and the migration doesn't result to a decrease in balance for any of these tokens.

To achieve this a loop inside a loop is executed where each token is the old strategy is checked against each token in the new strategy. This is because we aren't sure where the token is placed in the new array.

If it can be assumed that the tokens are placed in the array in an ascending order of their addresses, we can bring down complexity to linear.

Note that we don't need to sort the arrays but we can assume they are already sorted (so the responsibility of having it in ascending order of address falls to the strategy contract).

Note that this assumption doesn't lead to a hack. In the worst case, migration won't be possible.

Recommendation: Assume these arrays ares sorted and rewrite the loop to account for it.

Balmy: Since we expect most strategies to have one token, and the max will probably be around 3 tokens, the cost shouldn't be too high. Specially since this is calculated during registration/update.

At the same time, the cost will be paid by the strategy's owner, not any user.

Cantina Managed: Acknowledged.

3.5 Informational

3.5.1 Consider passing in token URI in the constructor in contract `EarnNFTDescriptor`

Severity: Informational

Context: EarnNFTDescriptor.sol#L9-L16

Description: The tokenURI is hardcoded and returns an empty string.

Recommendation: Consider passing in token URI in the constructor so third party such as opensea and query the nft data.

Balmy: Fixed in PR 69.

Cantina Managed: Fixed.

3.5.2 Loss of gas revenue in blast network

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The protocol is intended to be deployed to blast. In blast network, the gas fee spent by user can be claimed by contract owner (see the blast building guides). Yet because all smart contracts do not set gas mode to claimable and implement a function to claim the gas, those gas revenues are lost.

Recommendation:

```
interface IBlast {
  // Note: the full interface for IBlast can be found below
  function configureClaimableGas() external;
  function claimAllGas(address contractAddress, address recipient) external returns (uint256);
}

contract MyContrac is Ownable {
  IBlast public constant BLAST = IBlast(0x4300000000000000000000000000000000000002);

  constructor() {
    // This sets the Gas Mode for MyContract to claimable
    BLAST.configureClaimableGas();
  }

  // Note: in production, you would likely want to restrict access to this
  function claimMyContractsGas() onlyOwner external {
    BLAST.claimAllGas(address(this), msg.sender);
  }
}
```

for all contracts that will be deployed to blast, set gas mode to claimable and implement a function to claim gas.

Balmy: We've decided not to implement this change. Reasons are the following:

1. We don't have the space

Contract	Runtime Size (B)	Initcode Size (B)	Runtime Margin (B)	Initcode Margin (B)
EarnVault	23,501	26,077	1,075	23,075
FirewalledEarnVault	24,501	27,158	75	21,994

2. We don't think our Earn product is something that will generate big gas expenditure. Yield earning tends to be a set and forget action, unlike trading.

Cantina Managed: Acknowledged.

3.5.3 Consider adding access control to event emission functions in `GuardianManager.sol`

Severity: Informational

Context: GuardianManager.sol#L101-L113

Description: These three functions that have no access control and anyone can trigger them function to trick off-chain event tracking services:

```
function rescueStarted(StrategyId strategyId) external {
  emit RescueStarted(strategyId);
}
/// @inheritdoc IGuardianManagerCore

function rescueCancelled(StrategyId strategyId) external {
  emit RescueCancelled(strategyId);
}
/// @inheritdoc IGuardianManagerCore

function rescueConfirmed(StrategyId strategyId) external {
  emit RescueConfirmed(strategyId);
}
```

Only the `ExternalGuardian` should be executing these three functions.

Recommendation: Add access control to ensure only `ExternalGuardian` can call these three functions to trigger events.

Balmy: Initially those were no-ops, but when we added the events, we forgot to add any access control, We are fixing that in PR 103.

Cantina Managed: Fixed.

3.5.4 Add more testing for Compound strategy and LIDO strategy

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Currently only the ERC4626 strategy is tested. The integration test for Compound strategy and LIDO strategy are missing.

Recommendation: Add end-to-end testing for Compound strategy and LIDO strategy.

Balmy: Implemented tests for Compound in PR 106, and for Lido in PR 112.

Now, when we added the test, we realized that it wouldn't work correctly (it's explained on the PR's description). So we refactored it to make it deployable via factory.

Cantina Managed: Fixed.

3.5.5 Revert when 0 assets are deposited into a strategy

Severity: Informational

Context: EarnVault.sol#L576-L581

Description: `_increasePosition()` first transfers `depositedAmount` to `strategy` contract and then calls `strategy.deposited()` to notify `strategy` that certain amount was deposited. This call returns `assetsDeposited` which is the actual amount `strategy` considers to be deposited. This can be different from `depositedAmount` for reasons like a fee cut.

`_increasePosition()` reverts if `depositedAmount == 0`, but doesn't do the same when `assetsDeposited == 0`. This case may happen if strategy considers some small amount as dust and returns 0.

However, the code is still protected since in this case, it'll later revert with `ZeroSharesDeposit()`

Recommendation: Explicitly revert if `assetsDeposited == 0`.

Balmy: You are right that we are sill protected by `ZeroSharesDeposit`. We think it makes sense to leave it as it is right now in the sense that:

- We will revert with `ZeroAmountDeposit` if it was a user error.
- We will revert with `ZeroSharesDeposit` if the deposits ends up generating 0 shares due to rounding or something else.

Cantina Managed: Acknowledged.

3.5.6 Define `MAX_UINT_151` as `2**151 - 1`

Severity: Informational

Context: CustomUintSizeChecks.sol#L12

Description: It would be more readable to define `MAX_UINT_151` as `2**151 - 1` for clarity. it'll be computed at compile time.

Recommendation: Define `MAX_UINT_151` as `2**151 - 1`.

Balmy: Fixed in PR 73.

Cantina Managed: Verified.

3.5.7 Reference to a memory array passed twice to the same function call

Severity: Informational

Context: EarnVault.sol#L594-L597

Description: It's safe for now, but need to be careful here. Memory arrays are references to a memory location. So in this case, `balanceBeforeUpdate` and `balanceAfterUpdate` point to the same underlying data. If, in future, `_updateAccounting()` modifies any one argument, the other argument also changes to the updated value (so an update to `balanceBeforeUpdate[i]` also updates `balanceAfterUpdate[i]`).

Recommendation: This issue is reported for awareness. You may want to note this internally.

Balmy: Added notes in PR 80.

Cantina Managed: Verified.

3.5.8 Incorrect note about reentrancy check

Severity: Informational

Context: ExternalGuardian.sol#L196-L200

Description: The note below is incorrect:

> we disable the reentrancy check because the strategy should make sure this function
>
> // is called only by the vault, which already has a re-entrancy check

`_guardian_withdraw()` is called from `_fees_underlying_withdraw()` in `BaseStrategy.sol` which is ultimately called from `ExternalFees.withdrawFees()`. It only checks that `msg.sender` has the correct role. So it isn't called by a vault.

Recommendation: Actually, the `reentrancy-no-eth` check we disabled doesn't make sense anymore, due to one of the refactors we implemented for the audit. So I just removed it (and the note) in PR 126.

Cantina Managed: Fixed.

3.5.9 Rename `_connector_deposit()`

Severity: Informational

Context: LidoSTETHConnector.sol#L63, BaseConnectorInstance.sol#L61

Description: We recommend renaming `_connector_deposit()` to `_connector_deposited()` as the tokens are already deposited in this strategy. Same for `deposit()` in `BaseConnectorInstance`.

Recommendation: Rename these functions.

Balmy: Fixed in PR 116.

Cantina Managed: Fixed.

3.5.10 Document that strategy shouldn't withdraw less than requested amount

Severity: Informational

Context: EarnVault.sol#L293-L315

Description: There is an assumption here that strategy will always let you withdraw the entire amount stored in `withdrawn` (either immediate or delayed). `balanceAfterUpdate` isn't considered in accounting for positions asset and calculating `newPositionsShares`:

```
uint256 newPositionShares = _updateAccountingForAsset({
  positionId: positionId,
  strategyId: strategyId,
  totalShares: totalShares,
  positionShares: positionShares,
  positionAssetBalanceBeforeUpdate: positionAssetBalanceBeforeUpdate,
  totalAssetsBeforeUpdate: balancesBeforeUpdate[0],
  updateAmount: updateAmounts[0],
  action: action
});
```

Thus, if strategy withdraws a smaller amount of position asset for whatever reason, this logic breaks.

Recommendation: Document that vault assumes strategy reverts if it can't withdraw or reduce vault's balance by the request amount.

Balmy: Documented in PR 75.

Cantina Managed: Verified.