MiloTruck

Epoch Island

Security Review Report

October, 2023

Table of Contents

Introduction

About MiloTruck

MiloTruck is an independent security researcher who specializes in smart contract audits. Having won multiple audit contests, he is currently one of the top wardens on Code4rena. He is also a Senior Auditor at Trust Security and Associate Security Researcher at Spearbit.

For security consulting, reach out to him on Twitter - *@milotruck*

Disclaimer

A smart contract security review **can never prove the complete absence of vulnerabilities**. Security reviews are a time, resource and expertise bound effort to find as many vulnerabilities as possible. However, they cannot guarantee the absolute security of the protocol in any way.

Executive Summary

About Epoch Island

Epoch Island aims to become a Network State for crypto builders.

This codebase consists of two contracts to facilitate the protocol's initial time offering for their native token, EPOCH.

Repository Details

Repository	https://github.com/Moai-Labs/vepoch-contracts https://github.com/Moai-Labs/upside-contracts
Commit Hash	29b5dda948e856908e57afa7c4ace0f682ecb5eb d45051b65801157f039f4a88d8118e7d5b307e21
Language	Solidity

Scope

- vepoch-contracts/contracts/Vepoch.sol
- upside-contracts/contracts/EpochUpsidePoolV1.sol

Issues Found

Severity	Count
High	0
Medium	3
Low	4
Informational	10

Findings

Summary

ID	Description	Severity
M-01	Forfeited reward calculation in `withdrawForfeit()` breaks for multiple withdrawals	Medium
M-02	Makers can avoid protocol fees when providing liquidity	Medium
M-03	Use `SafeERC20` to handle token transfers	Medium
L-01	Funds might be stuck for tokens where `transfer()` reverts when `amount > uint96`	Low
L-02	Attackers can leverage flash loans to temporarily gain a large vEPOCH balance	Low
L-03	Violation of Checks-Effects-Interaction pattern	Low
L-04	Users can accidentally mint 0 vEPOCH when calling `deposit()`	Low
I-01	`depositForfeitAddress` is unused	Informational
I-02	Use `days` for time constants to improve readability	Informational
I-03	Logic in `extendDeposit()` can be simplified	Informational
I-04	Override `_beforeTokenTransfer()` instead	Informational
I-05	Minor refactor in `transferDepositOwnership()`	Informational
I-06	Redundant constructor in `EpochUpsidePoolV1.sol`	Informational
I-07	Refactor `claimYield()`	Informational
I-08	Gas savings in `withdraw()`	Informational
I-09	Gas savings in `deposit()`	Informational
I-10	Gas savings in `supply()`	Informational

Medium Severity Findings

M-01: Forfeited reward calculation in `withdrawForfeit()` breaks for multiple withdrawals

Bug Description

In `withdrawForfeit()`, the amount of rewards forfeited from withdrawing early is calculated as such:

Vepoch.sol#L186-L197

```
        uint256 forfeitReward = ((earned[_depositId] + rewardTokensClaimed[_depositId]) * percentage) / 1e18;

        // IF number of EPOCH to return is GREATER than pending unclaimed rewards
        // Transfer the excess from the user's wallet
        if(forfeitReward > earned[_depositId]) {
            // Calculate diff and transfer this many tokens from the user
            rewardToken.transferFrom(msg.sender, address(this), forfeitReward - earned[_depositId]);

            // Since the user didn't have enough earned and had to transfer tokens
            // This means we can set this to 0
            earned[_depositId] = 0;
        } else {
```

As seen from above, it is the withdrawal percentage multiplied by the deposit's unclaimed rewards (`earned`) + his claimed rewards (`rewardTokensClaimed`).

Afterwards, if `forfeitRewards` is larger than the deposit's unclaimed rewards, the caller is forced to transfer the difference into the contract and `earned` is set to 0.

However, the function does not subtract `forfeitReward - earned[_depositId]` from `rewardTokensClaimed[_depositId]`. This makes forfeited rewards calculation unfair when `withdrawForfeit()` is called multiple times.

First, consider a scenario where a user has not withdrawn any of his rewards:

- Assume a deposit is as follows:
    - `depositTokenBalance` is 1000 tokens
    - `earned[_depositId] = 1000e18`
    - `rewardTokensClaimed[_depositId] = 0`
- User withdraws 250 tokens:
    - `percentage` is 25%
    - `forfeitReward` is 25% of `1000e18 + 0`, which is `250e18`
    - `earned[_depositId]` becomes `750e18`
- User withdraws another 250 tokens:
    - `percentage` is 33.3% (250 / 750)
    - `forfeitReward` is 33.3% of `0 + 750e18`, which is `250e18`
    - `earned[_depositId]` becomes `500e18`
- In total, the user lost `500e18` reward tokens

Now, compare this to a user who has withdrawn all his rewards:

- Assume a deposit is as follows:
  - `depositTokenBalance` is 1000 tokens
  - `earned[_depositId] = 0`
  - `rewardTokensClaimed[_depositId] = 1000e18`
- User withdraws 250 tokens:
  - `percentage` is 25%
  - `forfeitReward` is 25% of `0 + 1000e18`, which is `250e18`
  - User transfers `250e18` reward tokens to the contract, since `forfeitReward > earned[_depositId]`
- User withdraws another 250 tokens:
  - `percentage` is 33.3% (250 / 750)
  - `forfeitReward` is 33.3% of `0 + 1000e18`, which is `333e18`
  - User transfers `333e18` reward tokens to the contract, since `forfeitReward > earned[_depositId]`
- In total, the user lost `583e18` reward tokens

Even though the amount of deposit withdrawn is the same in both scenarios, the user loses more reward tokens in the second one. This is because `rewardTokensClaimed[_depositId]` does not decrease, as mentioned above.

Impact

If users call `withdrawForfeit()` more than once after withdrawing a portion of their rewards, they will incorrectly forfeit more rewards.

Recommended Mitigation

Whenever a user transfers reward tokens when calling `withdrawForfeit()`, subtract the transferred amount from `rewardTokensClaimed`:

Vepoch.sol#L190-L197

```
        if(forfeitReward > earned[_depositId]) {
            // Calculate diff and transfer this many tokens from the user
            rewardToken.transferFrom(msg.sender, address(this), forfeitReward - earned[_depositId]);
+           rewardTokensClaimed[_depositId] -= forfeitReward - earned[_depositId];

            // Since the user didn't have enough earned and had to transfer tokens
            // This means we can set this to 0
            earned[_depositId] = 0;
        } else {
```

Team Response

Fixed in commit `3f95022`.

M-02: Makers can avoid protocol fees when providing liquidity

Bug Description

If makers wish to allow swaps with no protocol fees, they can set the `startDate` of their `LPPosition` to `0`. This causes `d.endDate - d.startDate` in `computeFee()` to be extremely large and `percentageFee` to be very small, thus the protocol fee will be minimal.

Note that this makes their own taker fee minimal as well.

Impact

Makers can intentionally create swaps with extremely small protocol fees, causing a loss of revenue for the protocol.

Recommended Mitigation

In `supply()`, check that `_startDate >= block.timestamp`.

Consider adding a `_startDate < _endDate` check as well, so that makers can't accidentally create "dead" positions where `take()` cannot be called at any point in time.

EpochUpsidePoolV1.sol#L59

```
        require(7501 > _feeBp, "FEE TOO HIGH");
+       require(_startDate >= block.timestamp, "START DATE < BLOCK.TIMESTAMP");
+       require(_startDate < _endDate, "START DATE >= END DATE");
```

Team Response

Fixed in commit `a6bffec`.

M-03: Use `SafeERC20` to handle token transfers

Bug Description

Both contracts use `transfer()` and `transferFrom()` to transfer tokens in many functions. However, this causes problems for two kinds of ERC20 tokens.

Missing Return Values

Some tokens do not return a `bool` (e.g. `USDT`, `BNB`, `OMG`) when `transfer()` is called, see here for a comprehensive (if somewhat outdated) list.

If such tokens are used, `transfer()` and `transferFrom()` will always revert when called. This is because the `IERC20` interface expects a bool to be returned:

IERC20.sol#L41

```
    function transfer(address to, uint256 value) external returns (bool);
```

IERC20.sol#L78

```
    function transferFrom(address from, address to, uint256 value) external returns (bool);
```

Thus, whenever `transfer()` or `transferFrom()` is called, Solidity will attempt to decode the return data into a `bool`. However, since such tokens do not return a `bool`, the decoding process will fail, causing the entire call to revert.

No Revert on Failure

Some tokens do not revert on failure, but instead return `false` (e.g. ZRX, EURS).

Since both contracts do not check the return value of `transfer()` or `transferFrom()`, it is possible for token transfers to silently fail without reverting.

Recommended Mitigation

Use `.safeTransfer()` instead of `.transfer()` in the following lines:

- Vepoch.sol#L108
- Vepoch.sol#L130
- Vepoch.sol#L210
- Vepoch.sol#L215
- Vepoch.sol#L234
- Vepoch.sol#L252
- EpochUpsidePoolV1.sol#L87
- EpochUpsidePoolV1.sol#L92
- EpochUpsidePoolV1.sol#L138
- EpochUpsidePoolV1.sol#L158
- EpochUpsidePoolV1.sol#L182
- EpochUpsidePoolV1.sol#L193

Use `.safeTransferFrom()` instead of `.transferFrom()` in the following lines:

- Vepoch.sol#L78
- Vepoch.sol#L146
- Vepoch.sol#L192

Team Response

Fixed in commit `3433f47` for `EpochUpsidePoolV1.sol`.

Acknowledged for `Vepoch.sol`.

Low Severity Findings

L-01: Funds might be stuck for tokens where `transfer()` reverts when `amount > uint96`

Bug Description

Some tokens, such as UNI and COMP, revert if the value passed to `transfer()` is larger than `uint96`. For example, the `transfer()` function for UNI is as shown:

Uni.sol#L400

```
function transfer(address dst, uint rawAmount) external returns (bool) {
    uint96 amount = safe96(rawAmount, "Uni::transfer: amount exceeds 96 bits");
    _transferTokens(msg.sender, dst, amount);
    return true;
}
```

For `EpochUpsidePoolV1.sol`, this becomes a problem in `claimProtocolFees()`, since it attempts to transfer the entire fee balance out in one call. For example:

- A maker places a huge `LPPosition` for UNI.
- The protocol fees for UNI accumulate until `protocolFeeBalances` exceeds `uint96`.
- When `claimProtocolFees()` is called to claim fees in UNI, it reverts. This is because the function calls `transfer()` with a balance larger than `uint96`.
- Therefore, all UNI fees are unclaimable forever.

For `Vepoch.sol`, this becomes a problem in `claimYield()` since it attempts to transfer a depositor's entire reward balance out in one call. If `earned[_depositId]` ever exceeds `uint96`, the depositor will never be able to claim yield as `claimYield()` will always revert.

Recommended Mitigation

In `claimProtocolFees()` and `claimYield()`, consider adding an `amount` parameter which allows the caller to specify the amount of tokens to transfer out in a single call.

Team Response

Fixed in commit `a6bffec` for `EpochUpsidePoolV1.sol`.

Acknowledged for `Vepoch.sol`.

L-02: Attackers can leverage flash loans to temporarily gain a large vEPOCH balance

Bug Description

`withdrawForfeit()` currently does not check if `_depositId` belongs to a deposit that was created in the same transaction. This makes it possible to abuse flash loans to temporarily gain a huge vEPOCH balance:

- Attacker takes out a flash loan of deposit token.
- Attacker calls `deposit()` with all his deposit tokens. This mints a huge amount of vEPOCH to the attacker.
- Attacker uses the vEPOCH balance to do whatever he wants.
- Attacker calls `withdrawForfeit()` to burn his vEPOCH and get deposit tokens in return.
- Attacker repays the flash loan.

This could be problematic if future contracts or functionality rely on a user's vEPOCH balance, such as checking `vEpoch.balanceOf(msg.sender))`.

Recommended Mitigation

Ensure that `withdrawForfeit()` cannot be called in the same transaction as `deposit()` for a single deposit. This can be achieved by ensuring `block.timestamp` is not the same:

Vepoch.sol#L172-L176

```
        // Ensure this function is only used for deposits where lock has not ended
        require(
            (d.depositTs + d.lockDuration) > block.timestamp,
            "DEPOSIT IS MATURED"
        );
+       require(d.depositTs != block.timestamp, "DEPOSIT IN SAME BLOCK");
```

Team Response

Fixed in commit `c65ad05`.

L-03: Violation of Checks-Effects-Interaction pattern

Bug Description

Throughout the contract, there are many tokens transfers performed before a state update, even though it is not necessary.

This violates the Checks-Effects-Interactions pattern, since external calls are performed before the contract's state is updated.

Should any token have user-controlled external calls (eg. ERC777 tokens have transfer hooks, which transfers execution control to the token sender), the contract might become vulnerable to reentrancy attacks.

For `EpochUpsidePoolV1.sol`, there is no restriction in `supply()` on what the upside/downside token address is. Therefore, the maker could even set the upside/downside token address to a malicious contract to gain a user-controlled external call.

Recommended Mitigation

For `EpochUpsidePoolV1.sol`, only perform token transfers at the end of `take()` and `untake()`.

For `Vepoch.sol`:

- Move L78 to the end of the `addRewardTokens()` function.
- Move L146 to the end of the `deposit()` function, just before the `return` statement.
- L192 should be after line 196, but still in the if-statement.
- L204 and L214 should be above line 190, since they should occur before any token transfer takes place.
- L234 should be right before the `return` statement.
- L252 should be at the end of the `withdraw()` function.

Team Response

Fixed in commit `3433f47` and `a788632`.

L-04: Users can accidentally mint 0 vEPOCH when calling `deposit()`

Bug Description

Since `calculateVeTokens()` uses division that rounds down, if a user calls `deposit()` with a small `_tokenAmount` and `_duration`, it is possible for `calculateVeTokens()` to round down to 0. This means that the depositor will get nothing in return for his deposit.

Recommended Mitigation

Consider checking that the amount of vEPOCH minted is not zero:

Vepoch.sol#L156

```diff
+       uint256 _mintAmount = calculateVeTokens(_tokenAmount, _duration);
+       require(_mintAmount != 0, "_tokenAmount TOO SMALL");
-       _mint(_behalfOf, calculateVeTokens(_tokenAmount, _duration));
+       _mint(_behalfOf, _mintAmount);
```

Team Response

Fixed in commit `840ead7`.

Informational Findings

I-01: `depositForfeitAddress` is unused

The `depositForfeitAddress` state variable is not used anywhere in the contract, and can be removed.

I-02: Use `days` for time constants to improve readability

Vepoch.sol#L19

```
-   uint256 public maxDepositDuration = 63072000;
+   uint256 public maxDepositDuration = 730 days;
```

Vepoch.sol#L325

```
-       require(_newMaxDepositDuration <= 315576000, "10 YEAR MAX");
+       require(_newMaxDepositDuration <= 3652.5 days, "10 YEAR MAX");
```

I-03: Logic in `extendDeposit()` can be simplified

`veTokenDiff` is equal to `calculateVeTokens(d.depositTokenBalance, _secondsToExtend)`, so there is no need to take the difference between the current and new balance:

Vepoch.sol#L267-L270

```
        // Determine how many more veTokens should be minted
-       uint256 currentVeTokenBalance = calculateVeTokens(d.depositTokenBalance, d.lockDuration);
-       uint256 newVeTokenBalance = calculateVeTokens(d.depositTokenBalance, d.lockDuration + _secondsToExtend);
-       uint256 veTokenDiff = newVeTokenBalance - currentVeTokenBalance;
+       uint256 veTokenDiff = calculateVeTokens(d.depositTokenBalance, _secondsToExtend);
```

I-04: Override `_beforeTokenTransfer()` instead

Instead of overriding `transferFrom()` and `transfer()`, use the `_beforeTokenTransfer()` hook to make tokens non-transferable. This can be done as such:

```
function _beforeTokenTransfer(address from, address to, uint256) internal override {
    if (from != address(0) && to != address(0)) {
        require(authorised[msg.sender], "NON TRANSFERABLE");
    }
}
```

With this, there is no need to override `transferFrom()` and `transfer()` individually.

I-05: Minor refactor in `transferDepositOwnership()`

Since `d` is a storage pointer, use `d.owner` below:

Vepoch.sol#L290-L291

```diff
         // Transfer ownership of deposit
-        deposits[_depositId].owner = _newOwner;
+        d.owner = _newOwner;
```

This helps to save a small amount of gas as well.

I-06: Redundant constructor in `EpochUpsidePoolV1.sol`

The constructor currently doesn't do anything. Consider setting `protocolFeeMaxBp` and `protocolFeeRecipientAddress` in the constructor, so that you don't have to call `setProtocolFee()` after deployment. Otherwise, remove the constructor.

I-07: Refactor `claimYield()`

Both `claimYield()` functions contain duplicated code. Consider refactoring the code to use a private function instead:

```
    function _claimYield(uint256 _depositId) private returns (uint256 reward) {
        // Ensure the claimer owns the specified deposit
        require(deposits[_depositId].owner == msg.sender, "NOT OWNER");
        _updateRewards(_depositId);

        reward = earned[_depositId];
        earned[_depositId] = 0;
        rewardTokensClaimed[_depositId] += reward;

        emit RewardClaimed(_depositId, reward);
    }

    // @notice Allows user to claim reward tokens earned for a given depositId
    function claimYield(uint256 _depositId) public {
        uint256 reward = _claimYield(_depositId);
        rewardToken.transfer(msg.sender, reward);
    }

    // @notice Allows user to claim reward tokens earned for one to many depositId's
    function claimYield(uint256[] calldata _depositIds) public {
        uint256 totalRewards;
        for(uint256 i = 0; i < _depositIds.length; i++) {
            uint256 totalRewards += _claimYield(_depositIds[i]);
        }
        rewardToken.transfer(msg.sender, totalRewards);
    }
```

I-08: Gas savings in `withdraw()`

In the `_tokenAmount == d.depositTokenBalance` body, use `_tokenAmount` instead of `d.depositTokenBalance` wherever possible to avoid reading from storage unnecessarily:

Vepoch.sol#L234

```
-           depositToken.transfer(msg.sender, d.depositTokenBalance);
+           depositToken.transfer(msg.sender, _tokenAmount);
```

I-09: Gas savings in `deposit()`

Cache `depositCount` in memory to avoid reading from storage multiple times:

Vepoch.sol#L143-L165

```
-   function deposit(uint256 _tokenAmount, uint32 _duration, address _behalfOf) external returns(uint256) {
+   function deposit(uint256 _tokenAmount, uint32 _duration, address _behalfOf) external returns(uint256 _depositCount) {

        require(_duration > 59 && _duration <= maxDepositDuration, "INVALID DURATION");
        depositToken.transferFrom(msg.sender, address(this), _tokenAmount);

-       depositCount += 1;
-       deposits[depositCount] = Deposit(
+       _depositCount = ++depositCount;
+       deposits[_depositCount] = Deposit(
            _behalfOf,
            uint32(block.timestamp),
            _duration,
            _tokenAmount
        );

        _mint(_behalfOf, calculateVeTokens(_tokenAmount, _duration));

        // Ensure this deposit is earning
-       _updateRewards(depositCount);
-       rewardStakingPower[depositCount] += calculateVeTokens(_tokenAmount, _duration);
+       _updateRewards(_depositCount);
+       rewardStakingPower[_depositCount] += calculateVeTokens(_tokenAmount, _duration);

-       emit Deposited(depositCount);
+       emit Deposited(_depositCount);

-       return depositCount;
+       return _depositCount;
    }
```

I-10: Gas savings in `supply()`

Cache `lpPositionCount` to avoid reading from storage multiple times:

EpochUpsidePoolV1.sol#L61-L73

```diff
+       uint256 _positionId = lpPositionCount++;
-       lpPositions[lpPositionCount] = LPPosition(
+       lpPositions[_positionId] = LPPosition(
            msg.sender,
            _feeBp,
            _startDate,
            _endDate,
            IERC20Metadata(_downsideToken),
            IERC20Metadata(_upsideToken),
            _upsideTokenAmount,
            _exchangeRate,
            0
        );
-       emit Supply(lpPositionCount);
-       lpPositionCount += 1;
+       emit Supply(_positionId);
```