Zellic

H20 vlPSDN

Smart Contract Security Assessment

May 15, 2023

Prepared for:

H20

Prepared by:

Katerina Belotskaia and Daniel Lu

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for H20 from April 24th to April 26th, 2023. During this engagement, Zellic reviewed H20 vlPSDN's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the distribution of rewards to users with locked tokens correct, especially when the contract owner has added/removed some reward tokens from the list?
- Are rewards calculated correctly?
- Can an attacker drain the vault and steal the locked or reward tokens?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

During this assessment, the limited scope prevented us from assessing potential interactions between the FutureRewardsVault contract and external parts of the project.

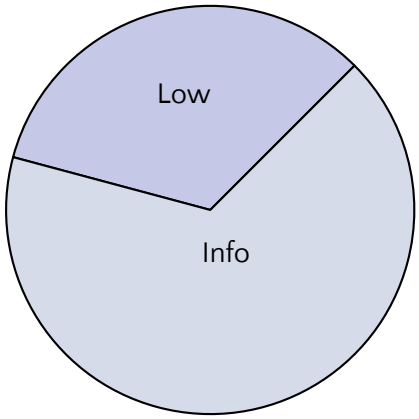
1.3 Results

During our assessment on the scoped H20 vlPSDN contracts, we discovered four findings. No critical issues were found. Of the four findings, two were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for H20's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	1
Informational	2



2 Introduction

2.1 About H20 vlPSDN

H20 vlPSDN is responsible for distributing rewards to users who lock their tokens in the contract. The contract owner sets the reward epochs and amounts, and at the end of each epoch, the contract distributes the rewards to the addresses that locked tokens during that epoch based on the amount of tokens they locked.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimiza-

tion, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

H20 vlPSDN Contracts

Repository	https://github.com/h2odata/h2o-vl-psdn
Version	h2o-vl-psdn: `79982f6b1ec4105e0b4777cf519ce089f60cf0a6`
Programs	• FutureRewardsVault • LockRewards
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of four person-days. The assessment was conducted over the course of three calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia, Engineer
kate@zellic.io

Daniel Lu, Engineer
daniel@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

April 25, 2023	Kick-off call
April 24, 2023	Start of primary review period
April 26, 2023	End of primary review period

3 Detailed Findings

3.1 The `_getAccount` function may return inaccurate information

- **Target**: LockRewards
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Informational

Description

The function returns the following information:

- `balance`, the amount of tokens deposited by the user
  - `lockEpochs`, the number of epochs for which the tokens are locked
  - `lastEpochPaid`, the last epoch for which the user has received rewards
  - `rewards`, an array of the rewards for each token

The function retrieves the first three values from the `accounts` mapping, while the last value is calculated in a for loop.

The loop iterates over the `rewardTokens` array, which contains the current list of reward tokens. However, since the `accounts[owner].rewards` mapping contains rewards for all tokens that the user has ever accrued and not claimed, if the user has accrued rewards for a token that is not in the current `rewardTokens` list, the function will not include it, resulting in an incomplete `rewards` list.

```
function _getAccount(address owner)
    internal
    view
    returns (uint256 balance, uint256 lockEpochs, uint256 lastEpochPaid,
    uint256[] memory rewards)
{
    rewards = new uint256[](rewardTokens.length);
    for (uint256 i = 0; i < rewardTokens.length;) {
        rewards[i] = accounts[owner].rewards[rewardTokens[i]];

        unchecked {
            ++i;
        }
    }
```

```
    return (accounts[owner].balance, accounts[owner].lockEpochs,
    accounts[owner].lastEpochPaid, rewards);
}
```

Impact

There are no security risks associated with this bug, but it could potentially cause confusion for users: the function may not accurately reflect the rewards that the user has accrued for tokens that are not currently in the reward tokens list.

Recommendations

We recommend modifying the for loop to iterate over the `accounts[owner].rewardTokens` array as shown below:

```
for (uint256 i = 0; i < accounts[owner].rewardTokens.length;) {
    address addr = accounts[owner].rewardTokens[i];
    uint256 reward = accounts[owner].rewards[addr];
    rewards[i] = reward;
    unchecked {
        ++i;
    }
}
```

Remediation

The issue has been fixed by H20 in commit 81f252c5.

3.2 Centralization risk: reward token recovery

- **Target**: LockRewards
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Informational

Description

- In the `recoverERC20` function, the owner can recover any ERC20 token excluding `lockToken`.
- In the `recoverERC721` function, the owner can recover any ERC721 token.

Impact

In the event that the `owner`'s private key is compromised, an attacker could potentially steal all reward tokens that have not yet been claimed by users by whitelisting a token and calling the `recoverERC20` function.

The `changeRecoverWhitelist` function does contain a check to prevent the owner from removing the governance token:

```
/**
 * @notice  Add or remove a token from recover whitelist,
 * cannot whitelist governance token
 * @dev Only contract owner are allowed.  Emits an event
 * allowing users to perceive the changes in contract rules.
 * The contract allows to whitelist the underlying tokens
 * (both lock token and rewards tokens).  This can be exploited
 * by the owner to remove all funds deposited from all users.
 * This is done bacause the owner is mean to be a multisig or
 * treasury wallet from a DAO
 * @param flag:  set true to allow recover
 */
function changeRecoverWhitelist(address tokenAddress, bool flag)
    external onlyOwner {
    if (tokenAddress == lockToken)
    revert CannotWhitelistLockedToken(lockToken);
    if (tokenAddress == rewardTokens[0])
    revert CannotWhitelistGovernanceToken(rewardTokens[0]);
    whitelistRecoverERC20[tokenAddress] = flag;
    emit ChangeERC20Whiltelist(tokenAddress, flag);
}
```

However, the check is ineffective because the owner can simply remove all tokens from `rewardTokens` using the `removeReward` function. This allows the owner to steal all reward funds.

Recommendations

- Use a multi-signature address wallet; this would prevent an attacker from causing economic damage if a private key were compromised.
- Set critical functions behind a timelock to catch malicious executions in the case of compromise.
- Prohibit withdrawal of reward tokens.

Remediation

H20 added a new role called `PAUSE_SETTER_ROLE` that is responsible for administering the `pause` and `unpause` functionality. Additionally, they have implemented the use of `TimeLockController` for ownership in commit 77d735f0.

3.3 Centralization risk: locked user funds

- **Target**: LockRewards
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The implementation of some protocol mechanics gives the owner the ability to lock user funds in the contract.

1. If deposit time is enforced, withdrawals require that the user has no more remaining epochs to wait.

```
/**
 * @notice Implements internal withdraw logic
 * @dev The withdraw is always done in name
 * of caller for caller
 * @param amount:  amount of tokens to withdraw
 */
function _withdraw(uint256 amount) internal {
    if (amount == 0 || accounts[msg.sender].balance < amount)
    revert InsufficientAmount();
    if (accounts[msg.sender].lockEpochs > 0 && enforceTime)
    revert FundsInLockPeriod(accounts[msg.sender].balance);

    IERC20(lockToken).safeTransfer(msg.sender, amount);
    totalAssets -= amount;
    accounts[msg.sender].balance -= amount;
    emit Withdrawn(msg.sender, amount);
}
```

2. Claiming rewards requires iteration over `account[msg.sender].rewardTokens`.

```
/**
 * @notice Implements internal claim rewards logic
 * @dev The claim is always done in name
 * of caller for caller
```

```
 * @return rewards of rewards transfer in token 1
 */
function _claim() internal returns (uint256[] memory rewards) {
    rewards = new uint256[](accounts[msg.sender].rewardTokens.length);

    for (uint256 i = 0; i < accounts[msg.sender].rewardTokens.length;) {
        address addr = accounts[msg.sender].rewardTokens[i];
        uint256 reward = accounts[msg.sender].rewards[addr];
        rewards[i] = reward;
        if (reward > 0) {
            accounts[msg.sender].rewards[addr] = 0;
            IERC20(addr).safeTransfer(msg.sender, reward);
            emit RewardPaid(msg.sender, addr, reward);
        }
        unchecked {
            ++i;
        }
    }

    accounts[msg.sender].rewardTokens = new address[](0);

    return rewards;
}
```

3. The owner has the ability to pause the protocol, which halts (among other functions) `withdraw`, `claimReward`, and `exit`.

Impact

1. The user's `lockEpochs` is only decremented as the owner continues to run epochs. If something goes wrong with the protocol or the owner was compromised, funds would be locked permanently.
2. The loop over `accounts[msg.sender].rewardTokens` could reach max gas. This might happen if the global `rewardTokens` were changed, each time causing new token addresses to be inserted into `accounts[msg.sender].rewardTokens` during accrual. It is unlikely for this to happen by accident (assuming minimal changes to reward tokens), but it would cause users to be unable to withdraw rewards.

3. If the protocol is paused, user funds become locked in the contract.

In general, these mechanics require the user to entrust the owner with the safety of their locked funds.

Recommendations

The protocol can be written against this trust requirement. First, users should have the ability to withdraw locked funds if epochs stop running — this would be accomplished by allowing withdrawals after a timelock at least as long as the lock duration multiplied by the epoch length.

Second, the gas issue could be mitigated by providing a function for claiming rewards for a given input token. That way, if the `rewardsToken` list becomes too large, users can simply withdraw them manually.

Lastly, the protocol should permit withdrawals (and possibly claims) even when paused. This ensures that users can trust in the safety of deposited funds.

Remediation

H20 implemented a mechanism for emergency withdrawals in commit 81f252c5. This allows users to eventually withdraw funds even if epochs halt. Additionally, the gas problems with `_claim` have been mitigated in commit a217e108. Finally, H20 acknowledges the centralization risks of the protocol's pause functionality.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Gas optimization for `_removeReward`

In LockRewards, the function `_removeReward` is used to remove a reward token from the `rewardTokens` list. However, the current implementation does so in an inefficient way: it copies all later elements one position to the left before popping the last element.

It can be rewritten to be more efficient, as shown below:

```
function _removeReward(uint256 index) internal {
    address addr = rewardTokens[index];
    rewardTokens[index] = rewardTokens[rewardTokens.length - 1];
    rewardTokens.pop();
    emit RemovedRewardToken(addr);
```

This optimization would provide significant gas savings. Additionally, this becomes a mechanism for cheaply removing tokens and would mitigate any possibility of max gas problems if `rewardTokens` became large.

Remediation

The issue has been fixed by H20 in commit 2c9ebd02.

4.2 Follow checks-effects-interactions pattern

We recommend following the checks-effects-interactions pattern in `LockRewards._withdraw` by changing the state of the contract before calling the external contract. Although we did not identify any reentrancy attacks, it is a best practice to prioritize security and prevent potential future attacks.

```
function _withdraw(uint256 amount) internal {
    if (amount == 0 || accounts[msg.sender].balance < amount)
    revert InsufficientAmount();
```

```
    if (accounts[msg.sender].lockEpochs > 0 && enforceTime)
    revert FundsInLockPeriod(accounts[msg.sender].balance);

    IERC20(lockToken).safeTransfer(msg.sender, amount);
    totalAssets -= amount;
    accounts[msg.sender].balance -= amount;
    emit Withdrawn(msg.sender, amount);
}
```

Remediation

The issue has been fixed by H20 in commit b605322d.

4.3 Input validation check

The `LockRewards.redeposit` function is used to increase a caller's deposit amount without extending the locking period. If the function is called when the caller has no deposit, the transaction will succeed, but the deposit will not be considered when calculating rewards.

Thus, it should only be called when the caller has an actual deposit and before the `lockEpochs` expire to increase the deposit and the reward. Additionally, calling the function with a zero amount is useless and should be prevented by adding a check that ensures the input `amount` is not zero and that the caller has a nonexpired deposit.

Remediation

The issue has been fixed by H20 in commit 046ab60b.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

> Our threat model was conducted on commit 79982f6b, which represents the codebase at a specific point in time. Therefore, it is possible that the absence of a particular test in our report may not reflect the current state of the test suite. We would like to note that during the remediation phase, H20 took significant steps to improve the code's quality and robustness. They added many additional test cases in commit fa48ec95, which is commendable and enhances the overall reliability of the system.

5.1 Module: FutureRewardsVault.sol

Function: `moveRewardsToLock(address[] tokens, uint256[] amounts)`

Transfers tokens from the contract to the `vlPsdn` address.

Inputs

- `tokens`
  - **Control**: Controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Decides what tokens are transferred into `vlPsdn`.
- `amounts`
  - **Control**: Controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Dictates the amount of each token transferred.

Branches and code coverage (including function calls)

Intended branches

- Should successfully transfer tokens if the caller has the `REWARDS_ROLE` role.
  ☐ Test coverage

Negative behavior

- Should revert if the caller does not have the `REWARDS_ROLE` role.
  ☐ Negative test

Function call analysis

- `moveRewardsToLock → IERC20(tokens[i]).safeTransfer`
  - **What is controllable?** Callee and arguments are controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `recoverERC20Token(address token)`

Allows the admin to withdraw any ERC20 tokens owned by the contract.

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Determines the token withdrawn.

Branches and code coverage (including function calls)

Intended branches

- Successfully transfers tokens if the caller has the `DEFAULT_ADMIN_ROLE` role.
  ☐ Test coverage

Negative behavior

- Reverts if the caller does not have the `DEFAULT_ADMIN_ROLE` role.
  ☐ Negative test

Function call analysis

- `recoverERC20Token → IERC20(token).safeTransfer`
  - **What is controllable?** Callee and arguments are controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.

- **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

5.2 Module: LockRewards.sol

Function: `changeEnforceTime(bool flag)`

Allows owner of contract to change `enforceTime` flag. If the flag is set to `false`, users will be able to withdraw their funds even if the `lockEpochs` is not expired. This is useful, in case of a major flaw in the contract, to prevent users from losing their funds.

Function: `changeRecoverWhitelist(address tokenAddress, bool flag)`

Allows owner to add or remove a token from recover whitelist. Cannot whitelist governance rewards tokens and locked tokens to prevent exploit of removing all funds deposited from all users by owner. Also if governance token is removed from `rewardTokens` array, it can be whitelisted.

Function: `claimReward()`

Allows user to receive its full claimable rewards. The `rewards` values for `msg.sender` and the `rewardTokens` list will be zeroed after successful transfer.

Branches and code coverage (including function calls)

Intended branches

- The `msg.sender` receives the proper amount of reward tokens.
  - ☐ Test coverage
- The `rewards` values for `msg.sender` and the `rewardTokens` list will be zeroed after successful transfer.
  - ☐ Test coverage

Negative behavior

- The `msg.sender` does not have claimable rewards.
  - ☐ Negative test
- The `msg.sender` cannot repeatedly claim the same reward after the first successful claim.
  - ☐ Negative test

Function call analysis

- `IERC20(addr).safeTransfer(msg.sender, reward);`
  - **External/Internal?** External.
  - **Argument control?** `reward`.
  - **Impact**: Transfer reward tokens from the contract to the caller.

Function: `deposit(uint256 amount)`

Allows any caller to deposit `lToken` tokens to this contract with `lockDuration` period, which is equal to 16 epochs and controlled by the owner of contract.

Inputs

- `amount`
  - **Validation**: The caller should be able to transfer this amount of `lToken` tokens to the contract.
  - **Impact**: The amount of deposited `lToken`.

Branches and code coverage (including function calls)

Intended branches

- The `lockEpochs` for `msg.sender` was increased by `lockDuration`.
  - ☐ Test coverage
- The `lToken` balance of contract was increased by `amount`.
  - ☐ Test coverage
- The `lToken` balance of `msg.sender` was decreased by `amount`.
  - ☐ Test coverage
- The balanceLocked for remaining epochs was increased by `amount`.
  - ☐ Test coverage
- totalLocked for remaining epochs was increased by `amount`.
  - ☐ Test coverage

Negative behavior

- Users do not have enough `lToken` tokens to deposit.
  - ☐ Negative test

Function call analysis

- `lToken.safeTransferFrom(msg.sender, address(this), amount)`
  - **External/Internal?** External.

- **Argument control?** `amount`.
- **Impact** Transfer `lToken` from the caller to this contract.

Function: `exit()`

Allows user to withdraw all their funds and receive all available rewards. If user funds are still locked, the transaction will revert.

Function: `pause()`

Allows owner to pause the contract. If contract is already paused, transaction will revert. Can only be called by the contract owner.

Function: `recoverERC20(address tokenAddress, uint256 tokenAmount)`

Allows owner to recover ERC20 sent by accident. All funds are only transferred to contract owner. To allow a withdraw, the token must be whitelisted by the owner using the `changeRecoverWhitelist`.

Function: `recoverERC721(address tokenAddress, uint256 tokenId)`

Allows owner of the contract to recover NFTs sent by accident. All funds are only transferred to contract owner.

Function: `redeposit(uint256 amount)`

Allows a caller to increase the deposit amount without increasing the locking period. In case a caller initiates first deposit over this function, the deposited funds will not be locked, will not bring a reward, and are immediately available for withdrawal. If executing this function until the `lockEpochs` has expired, then global `totalLocked` and `balanceLocked[msg.sender]` will be recalculated for the remaining number of `lockEpochs`. If the current epoch `isSet == true`, then funds will be added to epochs starting from the next epoch, otherwise from the current epoch. If the user calls `redeposit` after the `lockEpochs` have passed (even if the reward has not been accrued before, the `updateReward` will be called and reset the `lockEpochs`), the funds will be deposited but will not be taken into account when calculating the reward and will not be locked. It makes sense to call this function only before the `lockEpochs` expire.

Inputs

- `amount`
  - **Validation**: The caller should be able to transfer this amount of `lToken` tokens to the contract.

- **Impact**: The amount of deposited `lToken`.

Branches and code coverage (including function calls)

Intended branches

- The `lockEpochs` for `msg.sender` was not changed.
  - ☐ Test coverage
- The `lToken` balance of the contract was increased by `amount`.
  - ☐ Test coverage

Negative behavior

- User initiates the first deposit.
  - ☐ Negative test
- The amount is zero.
  - ☐ Negative test

Function call analysis

- `lToken.safeTransferFrom(msg.sender, address(this), amount)`
  - **External/Internal?** External.
  - **Argument control?** `amount`.
  - **Impact** Transfer `lToken` from the caller to this contract.

Function: `removeReward(address rewardToken)`

Allows owner to remove token from rewards. If reward token does not exist, transaction will revert. Removing a token from the list will not affect the rewards that have already been accrued. The users will be able to receive a reward with a removed token for the past epochs, even if the reward has not been withdrawn.

Function: `setLockDuration(uint256 duration)`

Allows owner to set lock period for users. If the same lock period is already set, transaction will revert. The `lockDuration` defines the number of epochs that a new deposit will lock for or the existing deposits will be able to relock for.

Inputs

- `duration`
  - **Validation**: Duration not equal to current `lockDuration`.
  - **Impact**: The new locking period for deposits.

Function: `setNextEpoch(uint256[] values)`

Allows caller with `EPOCH_SETTER_ROLE` to set a new epoch with default duration and `epochStart` equal to `now`. The amount needed of tokens should be transferred before calling `setNextEpoch`. If balance difference between current and unclaimed amount is less than provided in the `values` array, the transaction will revert.

Inputs

- `values`
  - **Validation**: `values.length` equal to `rewardTokens.length` — the difference between current and unclaimed amounts is greater or equal to `values[i]`.
  - **Impact**: The value of rewards for subsequent distribution among users — which funds will be locked for the next epoch.

Function: `setNextEpoch(uint256[] values, uint256 epochDurationInDays)`

Allows caller with `EPOCH_SETTER_ROLE` to set a new next epoch. The amount needed of tokens should be transferred before calling `setNextEpoch`. Can set `epochDurationInDays` different from default value.

Function: `setNextEpoch(uint256[] values, uint256 epochDurationInDays, uint256 epochStart)`

Allows caller with `EPOCH_SETTER_ROLE` to set a new next epoch. The amount needed of tokens should be transferred before calling `setNextEpoch`. Can set `epochDurationInDays` and `epochStart` different from default values.

Inputs

- `values`
  - **Validation**: `values.length` is equal to rewardTokens.length — the difference between current and unclaimed amount is greater or equal to values[i].
  - **Impact**: The value of rewards for subsequent distribution among users — which funds will be locked for the next epoch.
- `epochDurationInDays`
  - **Validation**: No checks.
  - **Impact**: The duration of the next epoch in days.
- `epochStart`
  - **Validation**: If `epochStart` is in the past, the transaction will revert.
  - **Impact**: The start date of the next epoch.

Function: `setReward(address rewardToken)`

Allows owner to set new reward token reward. If reward token is already on the list or is a token that is locked, transaction will revert. The reward token is the token that will be used to reward users for their deposits. The users will not receive a reward with a new token for the past epochs, even if the reward has not been withdrawn.

Inputs

- `rewardToken`
  - **Validation**: `rewardToken` is not equal to `lockToken`; `rewardToken` is not equal to any of the `rewardTokens`.
  - **Impact**: The new reward token.

Function: `unpause()`

Allows owner to unpause the contract. If contract is already unpaused, transaction will revert. Can only be called by the contract owner.

Function: `updateAccount()`

Function allows to trigger `updateEpoch` and `updateReward` logic without executing additional functionality and returning the current account state.

Function: `withdraw(uint256 amount)`

Allows user to withdraw deposited funds after the `lockEpochs` has expired or if `enforceTime` is false. The global `totalLocked` and `balanceLocked[msg.sender]` will be decreased by amount value.

Inputs

- `amount`
  - **Validation**: The `accounts[msg.sender].balance` should be greater or equal than the amount.
  - **Impact**: The amount of withdrawn `lToken`.

Branches and code coverage (including function calls)

Intended branches

- The `lockEpochs` for `msg.sender` is zero.
  - ☐ Test coverage
- The amount is less than or equal to `accounts[msg.sender].balance`.

- ☐ Test coverage

Negative behavior

- The amount is zero.
  - ☐ Negative test
- The amount is greater than `accounts[msg.sender].balance`.
  - ☐ Negative test
- The `lockEpochs` for `msg.sender` is greater than zero.
  - ☐ Negative test

Function call analysis

- `IERC20(lockToken).safeTransfer(msg.sender, amount)`
  - **External/Internal?** External.
  - **Argument control?** `amount`.
  - **Impact** Transfer `lToken` from the contract to the caller.

6 Audit Results

At the time of our audit, the code was not deployed to mainnet EVM.

During our audit, we discovered four findings. Of these, two were low risk and two were suggestions (informational). H20 acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.