Zellic

Rainmaker

Smart Contract Security Assessment

May 24, 2023

Prepared for:

Carlos Reyes Stoneham

Rainmaker

Prepared by:

Daniel Lu and Sina Pilehchiha

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 Rainmaker from May 17th to May 18th, 2023. During this engagement, Zellic reviewed Rainmaker'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:

- Does the bookkeeping accurately distribute rewards?
- Are there vulnerabilities that may result in loss of funds?
- What centralization risks exist in the protocol?

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 complexity of dependency mechanics (specifically Definitive vaults) limited the extent to which we could assess external interactions.

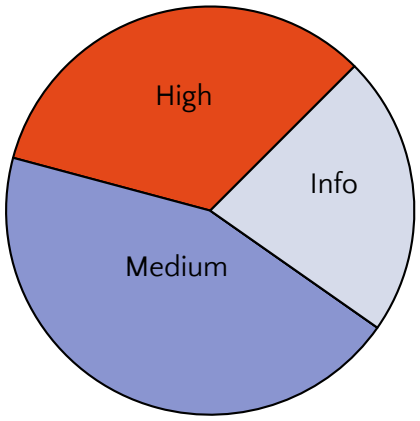
1.3 Results

During our assessment on the scoped Rainmaker contracts, we discovered nine findings. No critical issues were found. Of the other findings, three were of high impact, four were of medium impact, and two were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	3
Medium	4
Low	0
Informational	2



2 Introduction

2.1 About Rainmaker

Rainmaker is an all-in-one DeFi platform that offers gasless transactions, portfolio management, and one-click entry to DeFi strategies using an account abstraction–enabled smart contract wallet secured by a 4FA MPC wallet.

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 optimization, 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:

Rainmaker Contracts

Repository	https://github.com/rainmakerresearch/contracts
Version	contracts: `c42d8bc43d5102477dad3681b7d8856f770c1aab`
Programs	• DefinitiveRewardToken • DefinitiveStakingManager
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of one person-day. The assessment was conducted over the course of two 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:

Daniel Lu, Engineer
daniel@zellic.io

Sina Pilehchiha, Engineer
sina@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

May 17, 2023	Start of primary review period
May 18, 2023	End of primary review period
TBD	Closing call

3 Detailed Findings

3.1 Protocol owner can drain pools

- **Target**: DefinitiveRewardToken, DefinitiveStakingManager
- **Category**: Business Logic
- **Severity**: Critical
- **Likelihood**: Low
- **Impact**: High

Description

Each staking manager has an associated token for accounting purposes. When a user accrues rewards, the corresponding tokens are minted. When a user withdraws tokens, the contract uses the sum of their deposits and their reward token balance. From the `withdraw` function,

```
// Withdraw from definitive vault and include reward token balances
uint256 underlyingAmount = withdrawFromDefinitive(
  _index,
  _lpTokenAmount + rewardTokenBalance
);

emit Withdraw(msg.sender, underlyingAmount);

// Transfer to user
IERC20 underlying = IERC20(underlyingTokenAddresses[_index]);
underlying.approve(msg.sender, underlyingAmount);
underlying.transfer(msg.sender, underlyingAmount);
```

The reward token is deployed separately by the owner, who uses the admin role to grant the corresponding staking manager the ability to mint and burn tokens.

Impact

This means that the owner retains the ability to arbitrarily mint and burn tokens. By granting the `MINTER_ROLE` to an account they control, the owner can

1. decrease the shares of other users and
2. increase their own shares.

At any time, the owner can mint a large amount of tokens for themselves and withdraw the entire `lpTokensStaked`. As a consequence, in the event of a key compromise, all users would be exposed to potential loss of funds. Additionally, this requires unnecessary trust in the owner, which might discourage use of the protocol.

Recommendations

We recommend deploying the token contract from the staking manager constructor and removing the owner's responsibility to grant roles. Alternatively, the ownership of the contract could be transferred to the staking manager.

Remediation

In commit 5a9a0e3f, Rainmaker fixed this issue by deploying the token contract directly from the staking manager constructor.

3.2 Extraneous approval during withdrawal

- **Target**: DefinitiveStakingManager
- **Category**: Coding Mistakes
- **Severity**: Critical
- **Likelihood**: Medium
- **Impact**: High

Description

At the end of the `withdraw` function, the tokens are transferred to the user:

```
// Transfer to user
IERC20 underlying = IERC20(underlyingTokenAddresses[_index]);
underlying.approve(msg.sender, underlyingAmount);
underlying.transfer(msg.sender, underlyingAmount);
```

However, because the transfer is done with `transfer` and not `transferFrom` or `safeTransferFrom`, the approval to the sender is not spent. Even after the payment, the user can still withdraw `underlyingAmount` of the token by calling `transferFrom` themselves.

Impact

Definitive has the ability to withdraw tokens into the staking manager; the `withdrawTo` function is guarded with `onlyWhitelisted`. Thus, although no explicit functionality of the staking manager will leave the contract holding any funds, Definitive is allowed to perform such withdrawals and cause funds to be left in the staking manager contract.

Further, future functionality may include the staking manager taking custody of tokens (such as fees) as well.

In these cases, the extra approval will allow any user to steal rewards or future fees held by the staking manager. This could also be performed maliciously by Definitive. For instance, those with the `ROLE_DEFINITIVE` on the underlying strategy might be able to drain the contract by

1. depositing and withdrawing funds to increase unspent approval on the token,
2. calling `withdrawTo` on the underlying vault to withdraw funds into the staking manager, and
3. calling `transferFrom` on the underlying token into their own account.

Recommendations

Remove the unnecessary call to `underlying.approve`.

Remediation

This issue has been acknowledged by Rainmaker, and a fix was implemented in commit 46249703.

3.3 The underlying vault admin can drain pools

- **Target**: DefinitiveStakingManager
- **Category**: Coding Mistakes
- **Severity**: Critical
- **Likelihood**: Low
- **Impact**: High

Description

In underlying Definitive pools, the deployer can configure permissions during deployment. Importantly, a specific account is granted the `DEFAULT_ADMIN_ROLE`. In `CoreAccessControl`,

```
constructor(CoreAccessControlConfig memory cfg) {
    // admin
    _setupRole(DEFAULT_ADMIN_ROLE, cfg.admin);

    // definitive admin
    _setupRole(ROLE_DEFINITIVE_ADMIN, cfg.definitiveAdmin);
    _setRoleAdmin(ROLE_DEFINITIVE_ADMIN, ROLE_DEFINITIVE_ADMIN);

    // definitive
    for (uint256 i = 0; i < cfg.definitive.length; i++) {
        _setupRole(ROLE_DEFINITIVE, cfg.definitive[i]);
    }
    _setRoleAdmin(ROLE_DEFINITIVE, ROLE_DEFINITIVE_ADMIN);

    // clients - implicit role admin is DEFAULT_ADMIN_ROLE
    for (uint256 i = 0; i < cfg.client.length; i++) {
        _setupRole(ROLE_CLIENT, cfg.client[i]);
    }
}
```

In OpenZeppelin's `AccessControl`, the user with `DEFAULT_ADMIN_ROLE` has the ability to manage other roles.

Impact

This means that after deployment, the deployer is able to grant `ROLE_CLIENT` to other accounts. This allows them to steal funds by

1. granting that role to an account they control and

2. using that account to freely withdraw from the vault.

This would expose all users to potential loss of funds if the admin were ever compromised. It also requires unnecessary trust from users, which may discourage use of the protocol.

Recommendations

We recommend that Rainmaker set a smart contract or the vault manager itself as the sole owner of the vault. This may look like a system for transferring ownership during deployment.

Remediation

Rainmaker added documentation in commit ac95d65e, indicating that this risk is mitigated by granting underlying pool ownership to the Rainmaker multisig.

3.4 Missing slippage limits allow front-running

- **Target**: DefinitiveStakingManager
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

During deposits and withdrawals, the staking manager interacts with the underlying vault for entry and exit. Those functions each have the parameter `minAmount`, which sets slippage limits for the staker actions. However, the `minAmount` is set to zero in all cases:

```
/**
 * @dev Withdraw tokens from Definitive vault end-to-end (exit + withdraw)
 */
function withdrawFromDefinitive(
  uint8 _index,
  uint256 lpTokens
) private returns (uint256) {
  IERC20 underlying = IERC20(underlyingTokenAddresses[_index]);

  // 1. Exit from the strategy via LP Tokens
  uint256 underlyingAmount = definitiveVault.exitOne(lpTokens, 0, _index);

  // 2. Withdraw from the vault
  definitiveVault.withdraw(underlyingAmount, address(underlying));

  return underlyingAmount;
}
```

Impact

These slippage limits are essential for mitigating front-running. Consider the `_processExitPoolTransfers` function in Balancer's PoolBalances contract:

```
/**
 * @dev Transfers `amountsOut` to `recipient`, checking that they are
    within their accepted limits, and pays
 * accumulated protocol swap fees from the Pool.
 *
```

```
 * Returns the Pool's final balances, which are the current `balances`
    minus `amountsOut` and fees paid
 * (`dueProtocolFeeAmounts`).
 */
function _processExitPoolTransfers(
    address payable recipient,
    PoolBalanceChange memory change,
    bytes32[] memory balances,
    uint256[] memory amountsOut,
    uint256[] memory dueProtocolFeeAmounts
) private returns (bytes32[] memory finalBalances) {
    finalBalances = new bytes32[](balances.length);
    for (uint256 i = 0; i < change.assets.length; ++i) {
        uint256 amountOut = amountsOut[i];
        _require(amountOut ≥ change.limits[i], Errors.EXIT_BELOW_MIN);
        ...
    }
    ...
}
```

If the `minAmount` (which becomes an element of `change.limits`) is set to zero, the slippage check does nothing. This leaves users vulnerable to front-running.

Recommendations

We recommend that the protocol provide users a way to specify `minAmount`.

Remediation

This issue has been acknowledged by Rainmaker, and a fix was implemented in commit 2c613c09.

3.5 Unenforced assumptions about Definitive behavior

- **Target**: DefinitiveRewardToken
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The staking manager makes some assumptions about underlying vault behavior that may not be true. For instance, it treats the LP token balance as increasing (except during withdrawals). However, Definitive is permitted to perform exits and decrease that balance.

Impact

Violation of these assumptions might cause user funds to become locked. In staking manager withdrawal functions, withdrawals are always accompanied by exits:

```
/**
 * @dev Withdraw tokens from Definitive vault end-to-end (exit + withdraw)
 */
function withdrawFromDefinitive(
  uint8 _index,
  uint256 lpTokens
) private returns (uint256) {
  IERC20 underlying = IERC20(underlyingTokenAddresses[_index]);

  // 1. Exit from the strategy via LP Tokens
  uint256 underlyingAmount = definitiveVault.exitOne(lpTokens, 0, _index);

  // 2. Withdraw from the vault
  definitiveVault.withdraw(underlyingAmount, address(underlying));

  return underlyingAmount;
}
```

Since exiting and withdrawing are done together, there is no way to withdraw funds that are unstaked.

Those with `ROLE_DEFINITIVE` have permissions to unstake funds into the underlying vault. Additionally, as mentioned in 3.2, they are able to withdraw vault funds into the staking manager too.

These situations will create unstaked funds that cannot be withdrawn.

Recommendations

As a mitigation, Rainmaker could provide users the ability to redeposit and reenter funds if they get stuck in either the underlying vault or the staking manager.

Additionally, the vault is not immune to losses: it is possible for unfavorable conditions to cause a net decrease in LP token balance. This may result in shares that cannot be withdrawn. Rainmaker should document such risks.

Remediation

Rainmaker added functionality for restaking such funds in commit 25188ee8.

3.6 Excessive owner responsibility creates deployment risks

- **Target**: DefinitiveRewardToken, DefinitiveStakingManager
- **Category**: Code Maturity
- **Severity**: Medium
- **Likelihood**: Low
- **Impact**: Medium

Description

Each staking manager, from construction, holds an array of underlying token addresses:

```
// Constructor
constructor(
  address[] memory _underlyingAddresses,
  address _rewardTokenAddress,
  address _definitiveVaultAddress
) {
  underlyingTokenAddresses = _underlyingAddresses;
  rewardToken = DefinitiveRewardToken(_rewardTokenAddress);
  definitiveVault = IDefinitiveVault(_definitiveVaultAddress);
}
```

The precise order and contents of this array are extremely important because in `depositIntoDefinitive`, the `amounts` array must correspond exactly to both `underlyingTokenAddresses` and the token addresses in the vault.

```
/**
 * @dev Deposit tokens into Definitive vault end-to-end (deposit + enter)
 * @return Staked amount (lpTokens)
 */
function depositIntoDefinitive(
  uint256 _underlyingAmount,
  uint8 _index
) private returns (uint256) {
  IERC20 underlying = IERC20(underlyingTokenAddresses[_index]);

  uint256[] memory amounts
    = new uint256[](underlyingTokenAddresses.length);
  amounts[_index] = _underlyingAmount;

  // 1. Approve vault to spend underlying
  underlying.approve(address(definitiveVault), _underlyingAmount);
```

```
    // 2. Deposit into the vault
    definitiveVault.deposit(amounts, underlyingTokenAddresses);

    // 3. Enter into the strategy using 0 as minAmountsOut to get standard
      slippage
    return definitiveVault.enter(amounts, 0);
}
```

This means that during deployment, the owner is responsible for ensuring that `_underlyingAddresses` matches the vault's `LP_UNDERLYING_TOKENS`.

Additionally, the owner needs to grant the staking manager the `MINTER_ROLE` in its corresponding reward token.

Impact

If the `underlyingTokenAddresses` array does not match `LP_UNDERLYING_TOKENS`, the protocol may experience incorrect accounting or broken functionality.

If the staking manager is not granted the required role, then deposits and withdrawals would eventually fail. Worse, if `MINTER_ROLE` on one token is mistakenly granted to multiple different staking managers, they could experience severe accounting issues and users may lose funds.

Recommendations

We recommend that the protocol determine the underlying token addresses from the given vault as a single source of truth. The second issue is mitigated by the recommendation in 3.1.

Remediation

Rainmaker fixed these risks in 32bfa1fa and the remediations for 3.1.

3.7 Staking manager may become locked

- **Target**: DefinitiveStakingManager
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Description

The underlying vaults contain functionality that allows Definitive to pause contracts and the vault admin to unpause them. In `BaseAccessControl`,

```
/**
 * @dev Inherited from CoreStopGuardian
 */
function enableStopGuardian() public override onlyAdmins {
    return _enableStopGuardian();
}

/**
 * @dev Inherited from CoreStopGuardian
 */
function disableStopGuardian() public override onlyClientAdmin {
    return _disableStopGuardian();
}
```

The `STOP_GUARDIAN_ENABLED` flag is checked on critical strategy functions.

Impact

This means that the admin of the underlying strategy has the responsibility to prevent funds from being locked. In some unfavorable events (such as private key loss or compromise), staking manager mechanics may break.

Recommendations

In addition to the recommendations in 3.3, we recommend providing users some control over this "unpause" functionality — for example, by creating a smart contract, or modifying the staking manager, to act as the admin and allow users to unpause the contract. In case some pauses are necessary, this might include reasonable timelocks.

Remediation

In commit 6abfbd3d, Rainmaker documented that the admin role will be held by a multisig to mitigate centralization risk.

3.8 Potential centralization risk from fee configuration

- **Target**: DefinitiveStakingManager
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: N/A

Description

Though the value is not yet used, the staking manager allows the owner to set `feePct`:

```
/**
 * @dev Set fees
 */
function setFees(uint256 _feePct) external onlyOwner {
  feePct = _feePct;
}
```

Impact

If future additions to the protocol do use `feePct`, the owner would have the ability to make fees arbitrarily high — even above 100%. In general, this requires unnecessary trust from users, which might discourage use of the protocol. In the case of key compromise, this would grant an attacker the ability to steal additional user funds.

Recommendations

We recommend adding a reasonable upper limit (that is at least below 100%) on `feePct` if it is ever used. Alternatively, Rainmaker could instead implement a timelock for such configuration upgrades to allow users time to react to adverse changes.

Remediation

Rainmaker removed this functionality in 1d606d40.

3.9 Withdrawal conditions can cause confusion

- **Target**: DefinitiveStakingManager
- **Category**: Coding Mistakes
- **Severity**: Informational
- **Likelihood**: N/A
- **Impact**: N/A

Description

The `withdraw` function in DefinitiveStakingManager begins as follows:

```
/**
 * @dev Withdraw tokens from the strategy
 */
function withdraw(uint256 _lpTokenAmount, uint8 _index) public {
  StrategyStaker storage staker = strategyStakers[msg.sender];
  require(_lpTokenAmount > 0, "Withdraw amount can't be zero");
  require(_index < underlyingTokenAddresses.length, "Invalid index");

  // Withdraw all if amount is greater than staked
  if (_lpTokenAmount > staker.amount) {
    _lpTokenAmount = staker.amount;
  }
```

Since `staker.amount` may be zero, the rest of this function may continue with no tokens withdrawn despite the check on `_lpTokenAmount`.

Impact

The withdrawal process may be confusing to users with zero balance.

Further, suppose a user has zero balance in the staking manager, but nonzero balance in reward tokens. In order to redeem those tokens, they would have to call `withdraw` with nonzero `_lpTokenAmount` despite their balance being zero; otherwise, the function would revert. This is unintuitive.

Recommendations

We recommend performing the check on the *amount being withdrawn*. That is, after `_lpTokenAmount` is compared against the user balance and added to the `rewardTokenBalance`.

More discussion of `withdraw`'s behavior concerning reward token balance is included

in 4.1. If the suggestion to relocate token redemption logic is taken, then we simply recommend moving the `_lpTokenAmount > 0` check to after it is compared against `staker.amount`.

Remediation

This issue has been acknowledged by Rainmaker, and a fix was implemented in commit 4b8af543.

4 Discussion

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

4.1 Withdrawal redeems all tokens

When a user performs a withdrawal, the staking manager will redeem all `rewardTokens` they own as well. This behavior might be confusing, especially because `_withdraw` takes `_lpTokenAmount` as a parameter. A user might expect `withdraw` to redeem `_lpTokenAmount` of total tokens; instead, it withdraws `_lpTokenAmount` of the user's balance along with their entire balance of `rewardTokens`.

```
// burn reward tokens and get token credits
uint256 rewardTokenBalance = rewardToken.balanceOf(msg.sender);
rewardToken.burn(msg.sender, rewardTokenBalance);
```

In general, it may be unfavorable to withdraw in only a single token. Due to the underlying pool mechanics, this may cause users to experience excessive slippage. We suggest removing this functionality from `withdraw` and implementing a separate function for redepositing reward tokens into the protocol.

Alternatively, Rainmaker could simply make this behavior clear to users through documentation.

4.2 Underlying vault requires verification

The security and trust model of a given staking manager is heavily dependent on the underlying Definitive vault. It is the responsibility of users to verify that a staking manager's `definitiveVault`

1. is indeed one of the strategies deployed by Definitive and
2. is properly configured.

In particular, there should be no other accounts except for the staking manager with `ROLE_CLIENT` on the underlying `definitiveVault`. Otherwise, those accounts may have the ability to steal user funds. We encourage Rainmaker to provide thorough documentation for users.

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.

5.1 Module: DefinitiveRewardToken.sol

Function: `burn(address from, uint256 amount)`

Gives those with `MINTER_ROLE` the ability to burn tokens.

Inputs

- `from`
  - **Control**: Fully controlled by minter.
  - **Constraints**: None.
  - **Impact**: Allows tokens to be burned from any address.
- `amount`
  - **Control**: Fully controlled by minter.
  - **Constraints**: None.
  - **Impact**: Allows any amount of tokens to be burned.

Branches and code coverage (including function calls)

Intended branches

- Reward tokens are properly burned when the caller has `MINTER_ROLE`.
  ☑ Test coverage

Negative behavior

- Reverts when the caller does not have `MINTER_ROLE`.
  ☐ Negative test

Function: `mint(address to, uint256 amount)`

Gives accounts with the `MINTER_ROLE` the ability to mint tokens.

Inputs

- `to`
  - **Control**: Fully controlled by minter.
  - **Constraints**: None.
  - **Impact**: Allows tokens to be minted to any address.
- `amount`
  - **Control**: Fully controlled by minter.
  - **Constraints**: None.
  - **Impact**: Allows any number of tokens to be minted.

Branches and code coverage (including function calls)

Intended branches

- Reward tokens are properly minted when the caller has `MINTER_ROLE`.
  ☑ Test coverage

Negative behavior

- Reverts when the caller does not have `MINTER_ROLE`.
  ☐ Negative test

5.2 Module: DefinitiveStakingManager.sol

Function: `deposit(uint256 _underlyingAmount, uint8 _index)`

Enables depositing tokens into the vault and updating depositor's stake details.

Inputs

- `_underlyingAmount`
  - **Control**: Full.
  - **Constraints**: `_underlyingAmount > 0`.
  - **Impact**: The amount of tokens to be deposited.
- `_index`
  - **Control**: Full.
  - **Constraints**: Needs to be inside the bounds of `underlyingTokenAddresses`.

- **Impact**: The underlying token.

Branches and code coverage (including function calls)

Intended branches

- Successfully deposits `_underlyingAmount` into the vault.
  ☑ Test coverage

Negative behavior

- Reverts when `_underlyingAmount` is equal to `0`.
  ☐ Negative test
- Reverts when `_index` is out of bounds.
  ☐ Negative test
- Reverts when caller has insufficient balance of the `_index` token.
  ☐ Negative test

Function call analysis

- `deposit` → `harvestRewards()`
  - **What is controllable?** Discarded.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `deposit` → `depositIntoDefinitive(_underlyingAmount, _index)`
  - **What is controllable?** Discarded.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Denotes an issue with the definitiveVault contract.

Function: `harvestRewards()`

Harvests accumulated rewards.

Branches and code coverage (including function calls)

Intended branches

- Successfully transfer the rewards to the caller.
  ☑ Test coverage

Function call analysis

- `harvestRewards → definitiveVault.getAmountStaked()`
  - **What is controllable?** Discarded.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Denoting an issue with the definitiveVault contract. Interferes with the correct computation of rewards to be harvested.
- `harvestRewards → updatePoolRewards(newRewards)`
  - **What is controllable?** Discarded.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `harvestRewards → rewardToken.mint(msg.sender, rewardsToHarvest)`
  - **What is controllable?** Discarded.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Denoting an issue with the rewardToken contract.

Function: `setFees(uint256 _feePct)`

Sets the fees.

Inputs

- `_feePct`
  - **Control**: Full.
  - **Constraints**: Recommended to have a reasonable upper bound.
  - **Impact**: Fee percentage.

Branches and code coverage (including function calls)

Intended branches

- Needs to be tested on whether it is in reasonable bounds.
  ☐ Test coverage

Negative behavior

- Causes potential unexpected economic issues when it is not set at a reasonable

value.
  - ☐ Negative test

Function: `withdrawAll(uint8 _index)`

Allows to withdraw all staked tokens.

Inputs

- `_index`
  - **Control**: Full.
  - **Constraints**: Needs to be inside the bounds of `underlyingTokenAddresses`.
  - **Impact**: The underlying token.

Branches and code coverage (including function calls)

Intended branches

- Successfully withdraws all staked tokens.
  - ☑ Test coverage

Negative behavior

- Reverts when `_index` is out of bounds.
  - ☐ Negative test
- Reverts when `strategyStakers[msg.sender].amount` is equal to `0`.
  - ☐ Negative test

Function: `withdraw(uint256 _lpTokenAmount, uint8 _index)`

Withdraws token amount.

Inputs

- `_lpTokenAmount`
  - **Control**: Full.
  - **Constraints**: `_lpTokenAmount > 0`.
  - **Impact**: Amount of tokens to be withdrawn.
- `_index`
  - **Control**: Full.
  - **Constraints**: Needs to be inside the bounds of `underlyingTokenAddresses`.
  - **Impact**: The underlying token.

Branches and code coverage (including function calls)

Intended branches

- Successfully withdraws token amount.
  ☑ Test coverage

Negative behavior

- Reverts when `_lpTokenAmount` is lower than `0`.
  ☐ Negative test
- Reverts when `_index` is out of bounds.
  ☐ Negative test

Function call analysis

- `withdraw → harvestRewards()`
  - **What is controllable?** Discarded.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `withdraw → rewardToken.burn(msg.sender, rewardTokenBalance);`
  - **What is controllable?** Discarded.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Denotes an issue with the rewardToken contract.
- `withdraw → withdrawFromDefinitive(_index, _lpTokenAmount + rewardTokenBalance)`
  - **What is controllable?** Discarded.
- **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Denotes an issue with the definitiveVault contract.

6 Audit Results

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

During our assessment on the scoped Rainmaker contracts, we discovered nine findings. No critical issues were found. Three were of high impact, four were of medium impact, and the remaining findings were informational in nature.

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.