Zellic

Definitive

Smart Contract Security Assessment

August 31, 2023

Prepared for:

Blake Arnold

Definitive

Prepared by:

Sina Pilehchiha and Mohit Sharma

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 Definitive from August 30th to August 31st, 2023. During this engagement, Zellic reviewed Definitive's code for security vulnerabilities, design issues, and general weaknesses in security posture.

> Following the conclusion of this audit, we were requested to assess commit ecf13c96. We did not identify any security concerns arising from the changes introduced in this commit.

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 vault correctly mint the number of shares the user is owed, and are the shares of the users calculated correctly?
- Could users always redeem their shares for the correct value of the underlying private vault?
- Could users reduce the value of other users' shares?
- Could an attacker access another users' shares?

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.

1.3 Results

During our assessment on the scoped Definitive contracts, there were no security vulnerabilities discovered.

Zellic recorded its notes and observations from the assessment for Definitive's benefit in the Discussion section (3).

Breakdown of Finding Impacts

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

2 Introduction

2.1 About Definitive

Definitive is a DeFi gateway for institutional clients, providing smart vaults for yield management, trading, and leverage.

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 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 codebase 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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (3) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3 Scope

The engagement involved a review of the following targets:

Definitive Contracts

Repository	https://github.com/DefinitiveCo/contracts
Version	contracts: `93fac328c0b3754a3e4d3f2a81dab5b799b5eb68`
Program	MultiUserLPStakingStrategy.sol
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 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:

Sina Pilehchiha, Engineer
sina@zellic.io

Mohit Sharma, Engineer
mohit@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

August 30, 2023	Kick-off call
August 30, 2023	Start of primary review period
August 31, 2023	End of primary review period

3 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

3.1 Exploitation scenarios considered

Here, we aim to provide a summary of the possible exploitation scenarios we considered for the provided codebase.

ERC-4626 inflation attack

Since the given contract functions as a vault, there is potential attack surface for ERC-4626 vulnerabilities such as the inflation attack. However, the contract implements its vault functionality by inheriting from the OpenZeppelin ERC4626 contract. The project is using V4.9 of the OpenZeppelin contract, which offers out-of-the-box protection from ERC-4626 inflation attacks by using virtual decimal offsets. This makes it safe against the standard ERC-4626 attack vectors.

Share accounting errors

The contract accomodates functionality for minting and burning shares according to user withdrawals and deposits. We considered the following scenarios for the accounting mechanism:

1. Exploiting share-price manipulation with flash loans
2. Encountering situations where some shares are non-redeemable.
3. Minting or double minting unaccounted shares
4. Burning incorrect shares after withdrawals
5. Stealing other users' shares

Token considerations

Since the underlying asset might be ERC-777 compliant, we checked for any possible impact reentrancy vulnerabilities might have. In conclusion, the following solution taken from the contract MultiUserLPStakingStrategy.sol behaves as documented and maintains valid state in the case of reentrancy via transfer hooks in ERC-777:

```
function _deposit(address caller, address receiver, uint256 assets,
    uint256 shares) internal override {
    // If _asset is ERC777, `transferFrom` can trigger a reentrancy BEFORE
    the transfer happens through the
    // `tokensToSend` hook. On the other hand, the `tokenReceived` hook,
    that is triggered after the transfer,
    // calls the vault, which is assumed not malicious.
    //
    // Conclusion: we need to do the transfer before we mint so that any
    reentrancy would happen before the
    // assets are transferred and before the shares are minted, which is a
    valid state.
    // slither-disable-next-line reentrancy-no-eth
    IERC20(asset()).safeTransferFrom(caller, address(this), assets);

    (uint256[] memory assetAmounts, address[] memory assetAddresses)
    = (new uint256[](1), new address[](1));
    (assetAmounts[0], assetAddresses[0]) = (assets, asset());

    IERC20(asset()).resetAndSafeIncreaseAllowance(address(this),
    address(VAULT), assets);
    VAULT.deposit(assetAmounts, assetAddresses);

    if ((shares = previewDeposit(assets)) == 0) {
        revert ZeroShares();
    }

    // Calculate shares AFTER adding liquidity and BEFORE minting
    // `previewDeposit` calculates using the amount staked before
    additional liquidity is added
    VAULT.stake(assets);

    _mint(receiver, shares);

    emit Deposit(caller, receiver, assets, shares);
}

function _withdraw(
    address caller,
    address receiver,
    address _owner,
```

```
    uint256 assets,
    uint256 shares
) internal override {
    if (caller ≠ _owner) {
        _spendAllowance(_owner, caller, shares);
    }

    // If _asset is ERC777, `transfer` can trigger a reentrancy AFTER the
    transfer happens through the
    // `tokensReceived` hook. On the other hand, the `tokensToSend` hook,
    that is triggered before the transfer,
    // calls the vault, which is assumed not malicious.
    //
    // Conclusion: we need to do the transfer after the burn so that any
    reentrancy would happen after the
    // shares are burned and after the assets are transferred, which is a
    valid state.
    _burn(_owner, shares);

    VAULT.unstake(assets);
    VAULT.withdraw(assets, asset());

    IERC20(asset()).safeTransfer(receiver, assets);

    emit Withdraw(caller, receiver, _owner, assets, shares);
}
```

One crucial observation is that similar precautions are not taken for the other tokens that the contract interacts with (such as those in `_underlyingAssets`); however, we have not flagged this as a security issue as no active exploitation scenario was identified during the audit.

4 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the 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.

4.1 Module: MultiUserLPStakingStrategy.sol

Function: `depositUnderlying(uint256[] amounts, uint256 minAmount, address receiver)`

This mints vault shares to the receiver by depositing multiple underlying asset amounts.

Inputs

- `amounts`
  - **Control**: Full.
  - **Constraints**: The `amounts` and `assetAddresses` must be of the same length.
  - **Impact**: The amounts of each underlying asset to deposit.
- `minAmount`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: The minimum amount of vault shares to mint from the deposit.
- `receiver`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid, nonzero address.
  - **Impact**: Address to receive vault shares.

Branches and code coverage (including function calls)

Intended branches

- Should mint shares when invoking `depositUnderlying()` with LP tokens.
  ☑ Test coverage

Negative behavior

- `depositUnderlying()` should revert when in safe harbor mode.
  ☑ Negative test

Function call analysis

- `rootFunction → previewDeposit(uint256)`
  - **What is controllable?** N/A.
  - **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: `deposit(uint256 assets, address receiver)`

This mints vault shares to the receiver by depositing exactly the amount of underlying tokens.

Inputs

- `assets`
  - **Control**: Full.
  - **Constraints**: The `assets` needs to be less than the maximum amount of the underlying asset that can be deposited into the vault for the receiver, through a deposit call (`assets < maxDeposit(receiver)`).
  - **Impact**: Assets to be deposited.
- `receiver`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid, nonzero address.
  - **Impact**: Address to receive vault shares.

Branches and code coverage (including function calls)

Intended branches

- Should successfully mint shares when depositing with LP tokens.
  ☑ Test coverage
- Should return the correct result for `previewDeposit()` with zero outstanding shares.
  ☑ Test coverage
- Should return the correct result for `previewDeposit()` when there are outstanding share.
  ☑ Test coverage

- Should return `MAX_UINT_256` for `maxDeposit()`.
  ☑ Test coverage

Negative behavior

- `deposit()` should revert when in safe harbor mode.
  ☑ Negative test
- `previewDeposit()` should not revert when in safe harbor mode.
  ☑ Negative test
- `maxDeposit()` should not revert when in safe harbor mode.
  ☑ Negative test

Function call analysis

- `rootFunction → maxDeposit(address)`
  - **What is controllable?** N/A.
  - **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.
- `rootFunction → previewDeposit(uint256)`
  - **What is controllable?** N/A.
  - **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: `initialize(ILPStakingStrategy _vault, string _name, string _symbol)`

This initializes the contract with vault details, token name, and symbol and sets underlying assets and their count.

Inputs

- `_vault`
  - **Control**: Full.
  - **Constraints**: Must be a valid contract address.
  - **Impact**: The vault contract address.
- `_name`
  - **Control**: Full.
  - **Constraints**: Must be a valid string.
  - **Impact**: The name for the vault share token.
- `_symbol`

  - **Control**: Full.
  - **Constraints**: Must be a valid string.
  - **Impact**: The symbol for the vault share token.

Branches and code coverage (including function calls)

Intended branches

- Successfully initialize a contract.
  ☑ Test coverage

Function: `mint(uint256 shares, address receiver)`

This mints exactly vault shares to the receiver by depositing an amount of underlying tokens.

Inputs

- `shares`
  - **Control**: Full.
  - **Constraints**: Cannot exceed the max mintable shares.
  - **Impact**: The number of vault shares that will be minted and transferred to the receiver.
- `receiver`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid, nonzero address.
  - **Impact**: Address to receive vault shares.

Branches and code coverage (including function calls)

Intended branches

- Should mint shares when minting with LP tokens.
  ☑ Test coverage
- Should return the correct result for `previewMint()` with zero outstanding shares.
  ☑ Test coverage
- Should return the correct result for `previewMint()` when there are outstanding shares.
  ☑ Test coverage
- Should return `MAX_UINT_256` for `maxMint()`.
  ☑ Test coverage

Negative behavior

- `mint()` should revert when in safe harbor mode.
  ☑ Negative test
- `previewMint()` should not revert when in safe harbor mode.
  ☑ Negative test
- `maxMint()` should not revert when in safe harbor mode.
  ☑ Negative test

Function call analysis

- `rootFunction → maxMint(address)`
  - **What is controllable?** N/A.
  - **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.
- `rootFunction → previewMint(uint256)`
  - **What is controllable?** N/A.
  - **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: `redeemOneUnderlying(uint256 shares, uint8 index, uint256 minAmount, address receiver, address _owner)`

This redeems a specific underlying asset for a given number of shares.

Inputs

- `shares`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the owner's vault-share balance.
  - **Impact**: The number of the owner's shares that will be burned.
- `index`
  - **Control**: Full.
  - **Constraints**: Must be less than the total number of underlying assets.
  - **Impact**: Determines which specific underlying asset will be redeemed.
- `minAmount`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Reverts if received amount would be less than `minAmount`.
- `receiver`

  - **Control**: Full.
  - **Constraints**: Needs to be a valid, nonzero address.
  - **Impact**: Address to receive underlying asset.
- `_owner`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid, nonzero address.
  - **Impact**: Accounts whose shares will be burned in exchange for the asset.

Branches and code coverage (including function calls)

Intended branches

- Should burn shares when invoking `redeemOneUnderlying()` with LP tokens.
  ☑ Test coverage

Negative behavior

- `redeemOneUnderlying()` should revert when in safe harbor mode.
  ☑ Negative test
- `previewRedeem()` should not revert when in safe harbor mode.
  ☑ Negative test
- `maxRedeem()` should not revert when in safe harbor mode.
  ☑ Negative test

Function call analysis

- `rootFunction → maxRedeem(address)`
  - **What is controllable?** N/A.
  - **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.
- `rootFunction → previewRedeem(uint256)`
  - **What is controllable?** N/A.
  - **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: `redeemUnderlying(uint256 shares, uint256[] minAmounts, address receiver, address _owner)`

This redeems multiple underlying assets for a given number of shares.

Inputs

- `shares`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the owner's vault-share balance.
  - **Impact**: The number of the owner's shares that will be burned.
- `minAmounts`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: `minAmounts`.
- `receiver`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid, nonzero address.
  - **Impact**: Address to receive underlying assets.
- `_owner`
  - **Control**: Full.
  - **Constraints**: Must have share balance greater than or equal to shares.
  - **Impact**: Accounts whose shares will be burned in exchange for the asset.

Branches and code coverage (including function calls)

Intended branches

- Should burn shares when invoking `redeemUnderlying()` with LP tokens.
  ☑ Test coverage

Negative behavior

- `redeemUnderlying()` should NOT revert when in safe harbor mode.
  ☑ Negative test
- `previewRedeem()` should not revert when in safe harbor mode.
  ☑ Negative test
- `maxRedeem()` should not revert when in safe harbor mode.
  ☑ Negative test

Function call analysis

- `rootFunction → maxRedeem(address)`
  - **What is controllable?** N/A.
  - **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.

- `rootFunction → previewRedeem(uint256)`
  - **What is controllable?** N/A.
  - **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: `redeem(uint256 shares, address receiver, address _owner)`

This burns exactly shares from the owner and sends assets of underlying tokens to the receiver.

Inputs

- `shares`
  - **Control**: Full.
  - **Constraints**: Cannot exceed the owner's vault-share balance.
  - **Impact**: The number of the owner's vault shares that will be burned.
- `receiver`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid, nonzero address.
  - **Impact**: Address to receive underlying tokens.
- `_owner`
  - **Control**: Full.
  - **Constraints**: Must have an existing vault-share balance >= the shares being redeemed.
  - **Impact**: Account whose vault shares will be redeemed and burned.

Branches and code coverage (including function calls)

Intended branches

- Should burn shares when redeeming with LP tokens.
  ☑ Test coverage
- Should return correct result for `previewRedeem()`.
  ☑ Test coverage
- Should return total LP tokens when calling `maxRedeem()`.
  ☑ Test coverage

Negative behavior

- `redeem()` should revert when in safe harbor mode.
  ☑ Negative test

- `previewRedeem()` should not revert when in safe harbor mode.
  ☑ Negative test
- `maxRedeem()` should not revert when in safe harbor mode.
  ☑ Negative test

Function call analysis

- `rootFunction → maxRedeem(address)`
  - **What is controllable?** N/A.
  - **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.
- `rootFunction → previewRedeem(uint256)`
  - **What is controllable?** N/A.
  - **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: `setSafeAssets(address[] _safeAssets)`

This updates the list of safe assets and broadcasts the changes with an event.

Inputs

- `_safeAssets`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Overwrites the existing `SAFE_ASSETS` array.

Branches and code coverage (including function calls)

Intended branches

- Successfully updates the list of safe assets.
  ☐ Test coverage

Function: `withdraw(uint256 assets, address receiver, address _owner)`

This burns shares from the owner and sends exactly assets of underlying tokens to the receiver.

Inputs

- `assets`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to the max withdrawable assets for the owner.
  - **Impact**: The amount of underlying assets that will be transferred to the receiver.
- `receiver`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid, nonzero address.
  - **Impact**: Address to receive assets.
- `_owner`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid, nonzero address.
  - **Impact**: Address of the owner.

Branches and code coverage (including function calls)

Intended branches

- Should burn shares when withdrawing with LP tokens.
  ☑ Test coverage
- Should return correct result for `previewWithdraw()` with zero outstanding shares.
  ☑ Test coverage
- Should return total LP tokens when calling `maxWithdraw()`.
  ☐ Test coverage

Negative behavior

- `withdraw()` should revert when in safe harbor mode.
  ☑ Negative test
- `previewWithdraw()` should not revert when in safe harbor mode.
  ☑ Negative test
- `maxWithdraw()` should not revert when in safe harbor mode.
  ☑ Negative test

Function call analysis

- `rootFunction → maxWithdraw(address)`
  - **What is controllable?** N/A.
  - **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.
- `rootFunction → previewWithdraw(uint256)`
  - **What is controllable?** N/A.
  - **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 Assessment Results

At the time of our assessment, the reviewed code was deployed to the Ethereum Mainnet.

During our assessment on the scoped Definitive contracts, there were no security vulnerabilities discovered.

5.1 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 version reviewed during 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.