Zellic

Definitive LLSD

Smart Contract Security Assessment

Sept 13, 2023

Prepared for:

Blake Arnold

Definitive

Prepared by:

Ayaz Mammadov and Junyi Wang

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 Sept 7th to Sept 8th, 2023. During this engagement, Zellic reviewed Definitive LLSD'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:

- Are replay attacks possible with the signed payload?
- Is the receiving and repaying of flash loans properly secured?
- Can a user overleverage their position?
- Is it possible for funds to be stuck in an underlying protocol?

1.2 Non-goals and Limitations

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

- Failure of the underlying AAVE pools
- Incorrect signatures generated by the Definitive backend
- Failures of the underlying liquid-staking tokens
- Key custody of the deposit signing key

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 LLSD contracts, we discovered one finding, which was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for Definitive's benefit in the Discussion section (4).

Breakdown of Finding Impacts

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

2 Introduction

2.1 About Definitive LLSD

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 (4) 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 LLSD Contracts

Repository	https://github.com/DefinitiveCo/contracts/
Version	contracts: `6c150e4c7752b222515d273e9ad8b834b7a0975a`
Programs	MultiUserLLSDStrategy LLSD_EthereumAaveV3Balancer_wstETH_WETH
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:

Ayaz Mammadov, Security Engineer
ayaz@zellic.io

Junyi Wang, Security Engineer
junyi@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 7, 2023	Kick-off call
September 7, 2023	Start of primary review period
September 8, 2023	End of primary review period

3 Detailed Findings

3.1 First depositor issue

- **Target**: MultiUserLLSDStrategy
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The first depositor issue happens due to the inflation of shares, and it is especially valid when there are no shares in the vault.

A depositor making the first deposit to the vault could be front-run by a malicious attacker. The attacker would directly deposit into the vault without receiving shares, then during the processing of the first deposit, the depositor receives zero shares due to truncation, but the assets are still pulled.

```
function _getSharesFromDepositedAmount(uint256 assets)
    internal view returns (uint256) {
    uint256 _totalAssets = totalAssets();
    uint256 totalAssetsBeforeDeposit = _totalAssets > assets
    ? _totalAssets - assets : 0;
    return assets.mulDiv(totalSupply() + 10 ** _decimalsOffset(),
    totalAssetsBeforeDeposit + 1, Math.Rounding.Down);
}
```

Impact

The vault receives assets but did not generate a share. That is when an attacker mints one share and redeems it, stealing the initial deposit that was front-run.

Recommendations

Ensure that there is some seed liquidity in the vault.

Remediation

The Definitive team confirmed that they will launch their vaults with some initial liquidity.

4 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.

4.1 Centralization risk

There are several aspects of the contracts that are centralized, which could be dangerous in case of a security compromise or interruption of service.

The contracts are all upgradable, which allows a compromised Definitive key to steal user funds by upgrading the contract to withdraw and deposit to an attack address.

If the Definitive signing service is ever out of service, users would be unable to withdraw their funds for the duration as a signature is required to withdraw.

4.2 No salts when calculating hashes

Currently, the hashes are calculated in the manner below. The hashes have no salts/domains; therefore, if a change were ever introduced that make `DepositParams` the same size as `RedeemParams`, the signature supplied for one would also be valid for the other. This is a detail that should be taken into consideration when changing said params.

```
function encodeDepositParams(DepositParams calldata depositParams)
    public pure returns (bytes32) {
    return keccak256(abi.encode(depositParams));
}

function encodeRedeemParams(RedeemParams calldata redeemParams)
    public pure returns (bytes32) {
    return keccak256(abi.encode(redeemParams));
}
```

4.3 Transfer-limit griefing

Currently, a transfer limit is in place to limit transfers per block (by default 1); this however opens up an avenue for DOS attacks. A user could DOS the protocol by repeatedly submitting a transaction until its deadline is over as there is no nonce system to prevent. However, this would be very costly to maintain and not profitable.

```
function _enforceTransferLimits() private {
    if (block.number ≠ _latestTransfersBlockNumber) {
        _latestTransfersBlockNumber = block.number;
        delete _transfersThisBlock;
    }
    _transfersThisBlock += 1;
    if (_transfersThisBlock > MAX_TRANSFERS_PER_BLOCK) {
        revert TransfersLimitExceeded();
    }
}
```

5 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.

5.1 Module: LLSD_EthereumAaveV3Balancer_wstETH_WETH.sol

Function: `enter(uint256 flashloanAmount, SwapPayload swapPayload, uint256 maxLTV)`

This enters a position by swapping the deposited coins for the staked coin, putting it down as collateral, then repaying the flash loan by borrowing using the collateral.

Inputs

- `flashloanAmount`
  - **Constraints**: Must be signed by Definitive.
  - **Impact**: The size of the flash loan taken from Balancer.
- `swapPayload`
  - **Constraints**: Must be signed by Definitive.
  - **Impact**: Determines which coins to swap into the staked coin.
- `maxLTV`
  - **Constraints**: Must be signed by Definitive.
  - **Impact**: Limits the loan-to-value ratio after the transaction.

Branches and code coverage

Intended branches

- If `flashloanAmount` is zero, enter directly without flashloan.

☑ Test coverage

- Get a flash loan and call enter.

  ☑ Test coverage

Negative behaviour

- Reverts if caller is not whitelisted.

  ☐ Negative test

- Reverts if maximum LTV is exceeded after entering.

  ☑ Negative test

Function: `exit(uint256 flashloanAmount, uint256 repayAmount, uint256 de collateralizeAmount, SwapPayload swapPayload, uint256 maxLTV)`

This is an underlying implementation to exit the position by initiating a flash loan then returning the borrowed asset.

Inputs

- `flashloanAmount`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount to flash loan.
- `repayAmount`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount of borrowed asset to return (remove debt).
- `decollateralizeAmount`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount of collateral to remove from the borrow pool.

- `swapPayload`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The payload for the swap (swap information, in this case probably wstETH <–> WETH/ETH)
- `maxLTV`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Unused.

Branches and code coverage (including function calls)

Intended branches

- Initiates a flash loan.
  - ☑ Test coverage
- Handles zero `flashloanAmount`.
  - ☑ Test coverage

Negative behavior

- Cannot be called by nonwhitelist senders.
  - ☑ Negative test
- Reverts in case of LTV being above limit.
  - ☐ Negative test
- Does not execute in case of Stop guard.
  - ☑ Negative test

Function call analysis

- `initiateFlashLoan(flashloanAmount, abi.encode(FlashLoanContextType.EXIT, abi.encode(ctx)))`

- **What is controllable?** `flashLoanAmount` and `ctx`.
- **What happens if it reverts, reenters, or does other unusual control flow?** Cannot reenter due to reentrancy guard.
- **If return value is controllable, how is it used and how can it go wrong**: Discarded.

Function: `_enterContinue(byte[] contextData)`

This is an internal function to perform the concrete actions of entering the position.

Inputs

- `contextData`
  - **Constraints**: Internal function — only called using data signed by Definitive.
  - **Impact**: Swaps will be executed and an amount will be borrowed to repay the flash loan based on this data.

Branches and code coverage

Intended branches

- Swaps if a nonzero amount of swaps are required.
  - ☑ Test coverage

Function: `_exitContinue(byte[] contextData)`

This is the continuation of the `exit` function — has to be separated because of the flash loan implementation.

Inputs

- `contextData`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The data sent from `exit` through the balancer flash loan.

Branches and code coverage (including function calls)

Intended branches

- Repays the debt and decollateralizes.

  ☑ Test coverage

- Swap is executed.

  ☑ Test coverage

Negative behavior

- Does not swap if there is no swap payload.

  ☐ Negative test

Function call analysis

- `_repay(context.repayAmount)`
  - **What is controllable?** Full.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Nothing.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `_decollateralize(context.decollateralizeAmount)`
  - **What is controllable?** Full.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Nothing.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `_swap(swapPayloads, STAKING_TOKEN())`
  - **What is controllable?** `swapPayload`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Nothing.
  - **If return value is controllable, how is it used and how can it go wrong**:

Discarded.

5.2 Module: MultiUserLLSDStrategy.sol

Function: `deposit(address receiver, DepositParams depositParams, byte[] depositParamsSignature)`

This deposits funds into the vault, with a signature from Definitive minting shares to a specific address.

Inputs

- `receiver`
  - **Constraints**: None. None are necessary as the funds come from the sender.
  - **Impact**: The funds are deposited under this account.
- `depositParams`
  - **Constraints**: None enforced here but must be signed by Definitive.
  - **Impact**: Controls what tokens and in what amounts to deposit into the strategy.
- `depositParamsSignature`
  - **Constraints**: Must be a matching signature to `depositParams`.
  - **Impact**: Used to verify `depositParams`.

Branches and code coverage

Intended branches

- The right number of shares is minted to the right address.
  - ☑ Test coverage

Negative behavior

- Reverts if `depositParams` is not properly signed.
  - ☑ Negative test
- Reverts if the deadline is passed.

- ☐ Negative test

- Reverts if in Safe Harbor Mode.
  - ☑ Negative test
- Reverts if TLV limit is not within limits after transaction.
  - ☑ Negative test
- Reverts if the change in ratio to shares is too large.
  - ☑ Negative test
- Reverts if the depositor does not have enough funds.
  - ☐ Negative test
- Reverts if there are too many transfers per block.
  - ☑ Negative test

Function: `redeem(address receiver, address _owner, RedeemParams redeemParams, byte[] redeemParamsSignature)`

This redeems strategy shares and exit position.

Inputs

- `receiver`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The receiver of the underlying assets.
- `_owner`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The owner of the shares (the person whose shares will be burned).
- `redeemParams`
  - **Control**: Full.

  - **Constraints**: None.
  - **Impact**: The params used in the redeem (the swap payload, how much to repay back in the borrow pool, amount to flash loan to pay that back).

- `redeemParamsSignature`
  - **Control**: Full.
  - **Constraints**: Must be signed by `_signatureVerificationSigner`.
  - **Impact**: The signature approved by Definitive, allowing the redeem.

Branches and code coverage (including function calls)

Intended branches

- The flash loan is used to close a looped/leveraged position in a borrowed pool (the borrowed asset is paid back).
  - ☑ Test coverage
- Shares are burned.
  - ☑ Test coverage

Negative behavior

- Cannot transfer more than transfer limit.
  - ☑ Negative test
- Cannot execute tx past deadline.
  - ☑ Negative test
- Cannot execute with a non-Definitve signature.
  - ☐ Negative test
- Cannot execute in Safe Harbor Mode.
  - ☑ Negative test

Function call analysis

- `ILLSDStrategy(VAULT).STAKED_TOKEN()`:

- **What is controllable?** Nothing.
- **What happens if it reverts, reenters, or does other unusual control flow?** OK.
- **If return value is controllable, how is it used and how can it go wrong**: Staked token.

- `ILLSDStrategy(VAULT).STAKING_TOKEN()`:
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** OK.
  - **If return value is controllable, how is it used and how can it go wrong**: Staking token.

- `tokensRemoved.addresses[i].balanceOf((VAULT))`:
  - **What is controllable?** `tokensRemoved`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Nothing.
  - **If return value is controllable, how is it used and how can it go wrong**: Calculating balances to calculate total return.

- `totalAssets()`
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Used for calculating asset diff.

- `ILLSDStrategy(VAULT).exit(redeemParams.exitCtx ... )`
  - **What is controllable?** Everything.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.

- `tokensRemoved.addresses[i].balanceOf((VAULT))`:
  - **What is controllable?** `tokensRemoved`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Nothing.
  - **If return value is controllable, how is it used and how can it go wrong**: Calculating balances to calculate total return.
- `_spendAllowance(owner, _msgSender(), redeemParams.shares)`
  - **What is controllable?** `owner` and `redeemParams.shares`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts mean that the spender does not have enough allowance.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `_burn(_owner, redeemParams.shares)`
  - **What is controllable?** `owner` and `redeemParams.shares`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Nothing.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `_burn(_owner, redeemParams.shares)`
  - **What is controllable?** `owner` and `redeemParams.shares`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `_withdrawAndTransfer(receiver, tokensRemoved)`
  - **What is controllable?** `receiver`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

- **If return value is controllable, how is it used and how can it go wrong**: Discarded.

- `totalAssets()`
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Used for calculating asset diff.

6 Assessment Results

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

During our assessment on the scoped Definitive LLSD contracts, we discovered one finding, which was of low impact. Definitive acknowledged the finding and implemented a fix.

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 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.