Zellic

February 20, 2025

Prepared for
neeel.eth
unhedged21
0xYuba
Plaza Finance

Prepared by
Jinseo Kim
Chongyu Lv
Zellic

Plaza

Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Plaza Finance from February 13th to February 14th, 2025. During this engagement, Zellic reviewed Plaza's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. 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 the interactions of the PreDeposit contract with Balancer V2 correct?
- Are there any vulnerabilities that could result in the loss of user funds?

1.3. 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

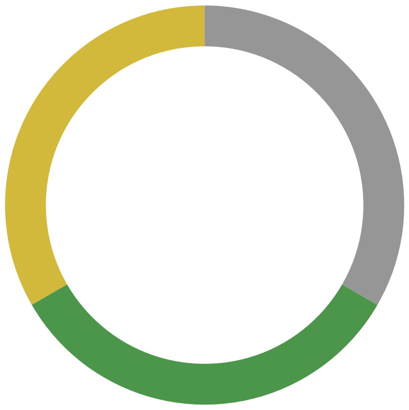
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Plaza contracts, we discovered three findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	1
🟩 Low	1
⬜ Informational	1



2. Introduction

2.1. About Plaza

Plaza Finance contributed the following description of Plaza:

> Plaza is a platform for programmable derivatives built as a set of Solidity smart contracts on Base. It offers two core products: bondETH and levETH, which are programmable derivatives of a pool of ETH liquid staking derivatives (LSTs) and liquid restaking derivatives (LRTs) such as wstETH. Users can deposit an underlying pool asset like wstETH and receive levETH or bondETH in return, which are represented as ERC20 tokens.
>
> BondETH and levETH represent splits of the total return of the underlying pool of ETH LSTs and LRTs, giving users access to composable profiles of risk and return that better suits their needs and investment style.

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, upgradability 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:

Plaza Contracts

Type	Solidity
Platform	EVM-compatible

Target	plaza-evm
Repository	https://github.com/Convexity-Research/plaza-evm ↗
Version	`0af9dc82c03171cf9fcacc306b890211033e94dd`
Programs	`PreDeposit.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of one and half person-days. The assessment was conducted by two consultants over the course of two calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Jinseo Kim
Engineer
jinseo@zellic.io

Chongyu Lv
Engineer
chongyu@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 13, 2025	Start of primary review period
February 14, 2025	End of primary review period

3. Detailed Findings

3.1. Weights are calculated through total balances of tokens

Target	PreDeposit		
Category	Business Logic	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The contract derives the weights of tokens from the ratio of balances of tokens:

```
function createPool(bytes32 salt)
    external nonReentrant whenNotPaused checkDepositEnded {
    // (...)

    // Determine the normalized weights of the tokens based on the balances of
    each token
    for (uint256 i = 0; i < nAllowedTokens; i++) {
      normalizedWeights[i] = amounts[i] * 1e18 / totalTokens;
    }

    // (...)
}
```

However, this effectively sets all relative prices of each token to 1, because Weighted Pool of Balancer V2 calculates the spot price between tokens through the following equation:

$$SP_i^o = \frac{\frac{B_i}{W_i}}{\frac{B_o}{W_o}}$$

Also, because the minimum weight of each token is 1% in Balancer V2, if the total deposited amount of a token is less than 1% of the total deposited amount of all tokens, it is impossible to initialize a pool:

```
library WeightedMath {
    // (...)

    // A minimum normalized weight imposes a maximum weight ratio. We need
    this due to limitations in the
    // implementation of the power function, as these ratios are often
    exponents.
    uint256 internal constant _MIN_WEIGHT = 0.01e18;

    // (...)
}
```

Impact

The tokens in the pool would be mispriced, providing an arbitrage opportunity to anyone who swaps on the created pool. Also, if the total deposited amount of a token is less than 1% of the total deposited amount of all tokens, it would be impossible to create a pool.

Recommendations

To remediate this issue, consider 1) calculating the weight of the pool in the way the initial spot prices are correctly determined and 2) ensuring all tokens are at least 1% deposited (by enforcing this condition on the `deposit` function or depositing at least 1% of the deposit cap for each token).

Remediation

This issue has been acknowledged by Plaza Finance, and fixes were implemented in the following commits:

- c34bd89d ↗
- ab5e7b32 ↗
- 0329956e ↗

3.2. The `_checkCap` function is missing a check

Target	PreDeposit		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `depositCap` state variable in the PreDeposit contract is used to represent the maximum deposit amount of the contract.

In the `PreDeposit::_checkCap` function, there is no check to ensure that `totalUserDepositValue > 0`.

```
function _checkCap(address[] memory tokens, uint256[] memory amounts)
    private view {
    uint256 totalUserDepositValue;
    for (uint256 i = 0; i < tokens.length; i++) {
      address token = tokens[i];
      uint256 price = balancerOracleAdapter.getOraclePrice(token, ETH);
      totalUserDepositValue += (amounts[i] * price) / 1e18;
    }

    if (totalUserDepositValue + currentPredepositTotal() > depositCap)
    revert DepositCapReached();
}
```

Impact

This may cause problems in some edge cases. Assuming `depositCap` is 1,000 Ether, when Chainlink price for `TOKEN1` is 0, users may deposit more `TOKEN1` than the `depositCap` limit.

Here is a proof-of-concept scenario:

1. First, user 1 deposits 99,999,900 Ether `TOKEN1` (Chainlink price for `TOKEN1` is 0).
2. Then, user 2 deposits 100 Ether `TOKEN2` (`TOKEN2` price is not 0).
3. Since user 2 deposits 100 Ether `TOKEN2`, the `_snapshotCapValue` in the `createPool` function is not 0, and the `createPool` function can be successfully executed.

```
// change TOKEN1_PRICE to 0 in PreDeposit.t.sol
function test_poc_CreatePool() public {
    console2.log("before deposit ,,, depositCap:  ", preDeposit.depositCap());
    vm.startPrank(user1);
    token1.approve(address(preDeposit), type(uint256).max);
    address[] memory tokens = new address[](1);
    tokens[0] = address(token1);
    uint256[] memory amounts = new uint256[](1);
    amounts[0] = 99999900 ether;
    preDeposit.deposit(tokens, amounts);
    vm.stopPrank();

    // user2 deposit 100 ether
    vm.startPrank(user2);
    token2.approve(address(preDeposit), type(uint256).max);
    address[] memory tokens2 = new address[](1);
    tokens2[0] = address(token2);
    uint256[] memory amounts2 = new uint256[](1);
    amounts2[0] = 100 ether;
    preDeposit.deposit(tokens2, amounts2);
    vm.stopPrank();

    vm.startPrank(governance);
    vm.warp(block.timestamp + 7 days + 1);

    preDeposit.setBondAndLeverageAmount(BOND_AMOUNT, LEVERAGE_AMOUNT);
    vm.warp(depositEndTime + 1 days); // After deposit period

    poolFactory.grantRole(poolFactory.POOL_ROLE(), address(preDeposit));

    bytes32 salt = bytes32("salt");
    vm.recordLogs();
    preDeposit.createPool(salt);

    vm.stopPrank();
}
```

Recommendations

Consider adding a check in the `_checkCap` function to ensure that `totalUserDepositValue` is greater than zero.

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit `cc43695d` ↗.

3.3. Missing length check

Target	PreDeposit		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the `_deposit` and `withdraw` functions, there is no check that the lengths of the `tokens` array and the `amounts` array are equal.

```
function _deposit(address[] memory tokens, uint256[] memory amounts,
    address recipient) private checkDepositStarted checkDepositNotEnded {
    _checkCap(tokens, amounts);

    for (uint256 i = 0; i < tokens.length; i++) {
      IERC20(tokens[i]).safeTransferFrom(msg.sender, address(this),
    amounts[i]);
      address token = tokens[i];
      uint256 amount = amounts[i];
      balances[recipient][token] += amount;
    }

    emit Deposited(recipient, tokens, amounts);
}

function withdraw(address[] memory tokens, uint256[] memory amounts) external
    nonReentrant whenNotPaused checkDepositStarted checkDepositNotEnded {
    for (uint256 i = 0; i < tokens.length; i++) {
      address token = tokens[i];
      uint256 amount = amounts[i];
      if (balances[msg.sender][token] < amount) revert InsufficientBalance();
      balances[msg.sender][token] -= amount;
      IERC20(token).safeTransfer(msg.sender, amount);
    }

    emit Withdrawn(msg.sender, tokens, amounts);
}
```

Impact

There is no security impact, and as such, this finding is reported as Informational. A mismatching length would just cause a revert. We report this with the purpose of improving the quality and consistency of the codebase.

Recommendations

Consider adding a check to ensure that the lengths of the `tokens` array and the `amounts` array are equal.

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit `a616df44` ↗.

4. Threat Model

This provides a full threat model description for various operations. As time permitted, we analyzed each operation handled by 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: PreDeposit.sol

Function: `claim()`

This function allows users to claim their share of bond and leverage tokens after pool creation.

Branches and code coverage

Intended branches

- ☑ Ensure that when two users deposit the same amount of tokens under the same conditions, they receive the same share of bonds after the pool is created.

Negative behavior

- ☑ Revert if claiming before deposit end.
- ☑ Revert if claiming before pool creation.
- ☑ Revert if claiming with zero balance.
- ☑ Revert if claiming twice.

Function: `createPool(bytes32 salt)`

This function creates a pool with the given salt.

Inputs

- `salt`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Salt value for creating a pool contract.

Branches and code coverage

Intended branches

- ☑ Calculate the total value of deposited tokens.
- ☑ Approve the deposited tokens for the pool to be created.
- ☑ Calculate the weights of the tokens.
- ☑ Create a pool.
- ☑ Store the snapshot of token prices.

Negative behavior

- ☑ No asset is deposited.
- ☑ Bond and leverage token amounts are not set.
- ☑ A pool is already created.

Function call analysis

- `this.currentPredepositTotal() -> this.balancerOracleAdapter.getOraclePrice(token, PreDeposit.ETH)`
    - **What is controllable?** None.
    - **If the return value is controllable, how is it used and how can it go wrong?** If the price from the oracle can be controlled, it may result in incorrect valuation of tokens.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.currentPredepositTotal() -> IERC20_1(token).balanceOf(address(this))`
    - **What is controllable?** None.
    - **If the return value is controllable, how is it used and how can it go wrong?** If the amount of a deposited token can be controlled, it may result in incorrect valuation of deposits.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `IERC20_1(this.allowedTokens[i]).balanceOf(address(this))`
    - **What is controllable?** None.
    - **If the return value is controllable, how is it used and how can it go wrong?** If the amount of a deposited token can be controlled, it may result in incorrect approval, reverting the pool creation or not affecting the business logic.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.balancerOracleAdapter.getOraclePrice(address(tokens[i]), PreDeposit.ETH)`

- **What is controllable?** None.
- **If the return value is controllable, how is it used and how can it go wrong?** If the amount of a deposited token can be controlled, it may result in incorrect valuation of deposits.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `IERC20_1(address(tokens[i])).approve(address(this.balancerVault), amounts[i])`
    - **What is controllable?** None.
    - **If the return value is controllable, how is it used and how can it go wrong?** None.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.balancerManagedPoolFactory.create(balancerPoolParams, balancerPoolSettingsParams, this.owner(), salt)`
    - **What is controllable?** Weights can be partially controlled as the amount of the token. Salt can be fully controlled by an unprivileged caller.
    - **If the return value is controllable, how is it used and how can it go wrong?** None.
    - **What happens if it reverts, reenters or does other unusual control flow?** The pool-creation logic may fail. This contract is not expected to reenter.
- `IManagedPool(address(balancerPoolToken)).getPoolId()`
    - **What is controllable?** None.
    - **If the return value is controllable, how is it used and how can it go wrong?** The contract may join an incorrect pool.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `balancerPoolToken.balanceOf(address(this))`
    - **What is controllable?** None.
    - **If the return value is controllable, how is it used and how can it go wrong?** Amount of reserve tokens may be incorrectly calculated, leading to failure of pool creation or locking of reserve tokens in the PreDeposit contract.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `balancerPoolToken.approve(address(this.factory), reserveAmount)`
    - **What is controllable?** None.
    - **If the return value is controllable, how is it used and how can it go wrong?** None.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.factory.createPool(this.params, reserveAmount, this.bondAmount,`

```
this.leverageAmount, this.bondName, this.bondSymbol, this.leverageName,
this.leverageSymbol, True)
```

- **What is controllable?** Bond and leverage amount can be controlled by the owner.
- **If the return value is controllable, how is it used and how can it go wrong?** None.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

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

The `withdraw` function is the core functionality for processing user withdrawal requests. It allows users to withdraw their previously deposited tokens from the contract.

Inputs

- `tokens`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of tokens to withdraw.
- `amounts`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of amounts to withdraw.

Branches and code coverage

Intended branches

- ☑ Withdraw multiple assets.

Negative behavior

- ☑ Revert if withdrawing after deposit.
- ☐ Revert if the length of the `tokens` array is not equal to the length of the `amounts` array.

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

This function is used to check if the deposit cap is reached.

Inputs

- `tokens`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of tokens to check.
- `amounts`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of amounts to withdraw.

Branches and code coverage

Intended branches

- ☑ Check if the deposit cap is reached.

Negative behavior

- ☑ Revert if `totalUserDepositValue` is 0.

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

The main function of the `_deposit` function is to transfer the tokens deposited by the user to the contract and update the user's deposit balance.

Inputs

- `tokens`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of tokens to deposit.
- `amounts`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of amounts to deposit.
- `recipient`
  - **Control**: Arbitrary.
  - **Constraints**: None.

- **Impact**: Beneficiary's address.

Branches and code coverage

Intended branches

- [x] Deposit multiple assets.

Negative behavior

- [x] Revert if depositing has not yet started.
- [x] Revert if depositing after end.
- [x] Revert if the deposit cap is reached.
- [ ] Revert if the length of the `tokens` array is not equal to the length of the `amounts` array.

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to Base Mainnet.

During our assessment on the scoped Plaza contracts, we discovered three findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining finding was informational in nature.

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.