Zellic

Prepared for
David Lee
Level

Prepared by
Juchang Lee
Sylvain Pelissier
Zellic

September 11, 2024

Points Farm

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 Level from September 3rd to September 4th, 2024. During this engagement, Zellic reviewed Points Farm'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 there any vulnerabilities that could result in the loss of user funds?
- Could an admin key compromise lead to loss of user funds?
- Is there insufficient access control for any critical functions?

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

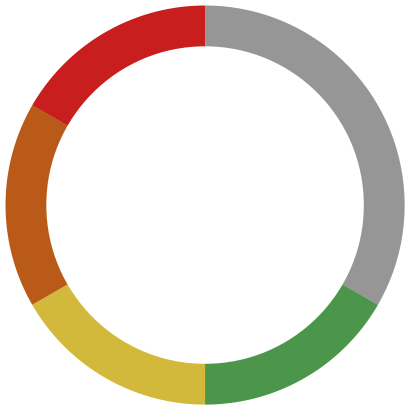
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Points Farm contracts, we discovered six findings. One critical issue was found. One was of high impact, one was of medium impact, one was of low impact, and the remaining findings were informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	1
🟧 High	1
🟨 Medium	1
🟩 Low	1
⬜ Informational	2



2. Introduction

2.1. About Points Farm

Level contributed the following description of Points Farm:

> Level is a stablecoin protocol powered by restaked dollar tokens like USDT and USDC.

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:

Points Farm Contracts

Type	Solidity
Platform	EVM-compatible

Target	points-farm
Repository	https://github.com/Level-Money/points-farm ↗
Version	`7329e6411ce2faeae831f83e2fb65f7a0e2088a9`
Programs	`LevelStakingPool`

2.4. Project Overview

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

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

The following consultants were engaged to conduct the assessment:

Juchang Lee
Engineer
lee@zellic.io

Sylvain Pelissier
Engineer
sylvain@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

September 3, 2024	Start of primary review period
September 4, 2024	End of primary review period

3. Detailed Findings

3.1. Signature replay allows unauthorized migrations

Target	LevelStakingPool		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Medium	**Impact**	Critical

Description

The `migrate` function in the contract enables users to convert their tokens into lvlUSD. For a migration to proceed, it must first be authorized by the `levelSigner` contract. To verify the migration, the contract generates a hash that includes the migrator's address, the signature's expiration date, the contract's address, and the current chain ID.

```
function migrate(
    address[] calldata _tokens,
    address _migratorContract,
    address _destination,
    uint256 _signatureExpiry,
    bytes calldata _authorizationSignatureFromLevel
) external {
    uint256[] memory _amounts = _migrateChecks(
        msg.sender,
        _tokens,
        _signatureExpiry,
        _migratorContract
    );

    bytes32 constructedHash = keccak256(
        abi.encodePacked(
            "\x19Ethereum Signed Message:\n32",
            keccak256(
                abi.encodePacked(
                    _migratorContract,
                    _signatureExpiry,
                    address(this),
                    block.chainid
                )
            )
        )
    );
//...
```

Then the signature provided as an argument is verified to be correct with respect to the `levelSigner` contract:

```
// verify that the migrator's address is signed in the authorization signature
    by the correct signer (levelSigner)
    if (
        !SignatureChecker.isValidSignatureNow(
            levelSigner,
            constructedHash,
            _authorizationSignatureFromLevel
        )
    ) {
        revert SignatureInvalid();
    }
```

However, there is no mechanism in place to prevent the same signature from being replayed to authorize other user migrations.

Impact

If another user obtains a valid signature from the `levelSigner` contract, they can replay the signature before the `_signatureExpiry` time, allowing them to migrate their tokens without proper authorization.

Recommendations

We recommend that this function include nonces to prevent replay attacks and add the sender's address to the hash, binding the signature to a specific user.

Remediation

This issue has been acknowledged by Level. Level will disable the migrate functionality and ask users to withdraw funds and convert them to lvlUSD tokens by other means.

3.2. No test suite

Target	LevelStakingPool		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

There is currently no test suite for this codebase.

Impact

When building a project with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch. For example, Finding 3.6. ↗ could have been discovered by a test.

The test suite for this project should be implemented with a large coverage to include all contract code, not just surface-level functions. It is important to test the invariants required for ensuring security. Testing the findings reported during this audit should be implemented to avoid regression in the future. Additionally, testing cross-contract function calls and transfers is recommended to ensure the desired functionality.

Recommendations

We recommend building a rigorous test suite to ensure that the system operates securely and as intended.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality *most likely* was not broken by your change to the code.

Remediation

This issue has been acknowledged by Level.

3.3. Tokens from previous migrations may be lost

Target	LevelStakingPool		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

Inside the `migrate` function, after verifying the signature, the `_migrate` function is called to update the user's balance and grant the migrator contract approval to transfer the user's tokens.

```
function _migrate(
     address _user,
     address _destination,
     address _migratorContract,
     address[] calldata _tokens,
     uint256[] memory _amounts
 ) internal {
     uint256 length = _tokens.length;
     //effects for-loop (state changes)
     for (uint256 i; i < length; ++i) {
         //if the balance has been already set to zero, then _tokens[i] is a
    duplicate of a previous token in the array
         if (balance[_tokens[i]][_user] == 0) revert DuplicateToken();

         balance[_tokens[i]][_user] = 0;
     }

     emit Migrate(
         ++eventId,
         _user,
         _tokens,
         _destination,
         _migratorContract,
         _amounts
     );

     //interactions for-loop (external calls)
     for (uint256 i; i < length; ++i) {
         IERC20(_tokens[i]).approve(_migratorContract, _amounts[i]);
     }
 //...
```

However, the `approve` function may decrease the allowance of the user, and if the previous allowance was not transferred, it may be lost since the user balance is set to zero between each migration.

The vulnerability comes from ERC-20's `_approve` function, which sets the spender's allowance without checking if the previous allowance was transferred or not.

Impact

Depending on the implementation of the migrator contract, the user may lose funds.

For example, if the user first migrates two tokens, then the contract will approve for two tokens. If the migrator does not transfer the tokens directly, and meanwhile the user wants to migrate one more token, then the allowance will be decreased to one by the `approve` function and the migrator contract will only be able to transfer a single token — not three, as intended.

Recommendations

The contract should use the function `safeDecreaseAllowance` instead of `approve` as defined in OpenZeppelin's SafeERC20 ↗ implementation.

Remediation

This issue has been acknowledged by Level. Level will disable the migrate functionality and ask users to withdraw funds and convert them to lvlUSD tokens by other means.

3.4. Uneffective wETH limit

Target	LevelStakingPool		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The maximum amount staked by a user is set by the mapping `tokenBalanceAllowList`, which is configured by the constructor of the function `setStakableAmount`. When a user wants to deposit tokens, the amount sent is checked to be less than the limit and the amount already staked by the `depositFor` function. However, to deposit wETH, the function `depositETHFor` has to be used by the user:

```
function depositETHFor(address _for) external payable whenNotPaused {
    if (msg.value == 0) revert DepositAmountCannotBeZero();
    if (_for == address(0)) revert CannotDepositForZeroAddress();
    if (tokenBalanceAllowList[WETH_ADDRESS] == 0)
    revert TokenNotAllowedForStaking();

    balance[WETH_ADDRESS][_for] += msg.value;
    emit Deposit(++eventId, _for, WETH_ADDRESS, msg.value);

    IWETH(WETH_ADDRESS).deposit{value: msg.value}();
}
```

In this function, the limit is not checked, the maximum value is only checked to be nonzero. Thus, the user is allowed to deposit any amount as soon as the limit is nonzero.

Impact

Since the balance limit is ineffective for wETH, the contract is not able to control the amount of wETH staked.

Recommendations

We recommend to check the amount of wETH deposited as it is done in the `depositFor` function.

Remediation

This issue has been acknowledged by Level. Level is considering not accepting wETH anymore by setting the stakable amount to zero.

3.5. ERC-20 approve function reverts on nonstandard token

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

Description

As explained in Finding 3.3. ↗, the function `_migrate` is called in order to update user balance and approve the migrator contract to transfer the user's token. For ERC20 tokens, the `approve` function must return a `bool` value. If not, the contract call to `approve` reverts. Thus, the migration may not work for non-ERC20 tokens like USDT.

Impact

According to the documentation ↗, a user should be able to restake USDT. However, USDT does not adhere to the ERC20 token standard. The USDT `approve` function does not return anything. Since the contract is expecting a boolean value but, for USDT, does not get anything, then the contract reverts, preventing the user from migrating their USDT. However, no funds would be locked, and the users can withdraw their funds whenever they want.

Recommendations

The contract should use the function `forceApprove` from SafeERC20 ↗, which was created specially to handle those cases of ERC20 and nonstandard tokens.

Remediation

This bug was reported by Level during the preaudit phase. Level will disable the migrate functionality and ask users to withdraw USDT and convert them to lvlUSD tokens by other means.

3.6. Owner of the contract can arbitrarily prevent deposits and migrations

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

Description

The owner of the contract has the ability to change the signer contract and the stakeable amount at any time via the `setLevelSigner` and `setStakableAmount` functions. If the owner changes the signer contract, previous signatures from the former signer will no longer be valid for verification by the contract. Additionally, altering the stakeable amount could prevent users from depositing tokens into the contract.

Impact

The owner is able to prevent a user from migrating their tokens by changing the `levelSigner` just before a user calls the `migrate` function. The owner can also prevent deposits by changing the stakeable amount. However, those griefing attacks do not result in a loss of funds since the user is able to withdraw their funds at any time. Only the gas paid is lost.

Recommendations

Consider making `setLevelSigner` and `setStakableAmount` callable only when the contract is paused.

Remediation

This issue has been acknowledged by Level.

4. Assessment Results

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

During our assessment on the scoped Points Farm contracts, we discovered six findings. One critical issue was found. One was of high impact, one was of medium impact, one was of low impact, and the remaining findings were informational in nature.

4.1. 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.