Zellic

Prepared for
Solera

Prepared by
Aaron Esau
Seunghyeon Kim
Zellic

March 19, 2025

Solera

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 Solera from March 13th to March 17th, 2025. During this engagement, Zellic reviewed Solera'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 all of the functions implemented as intended?
- Will the calculate-borrow-amount functions overflow/underflow or revert unexpectedly?
- If a malicious token in the swap path were malicious, what could it do?
- Are there situations in which reentrancy through WrappedTokenGatewayV3 (functions missing reentrancy guards) could be exploited?
- Are there any ways to drain the tokens from the contract mid-transaction?

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.

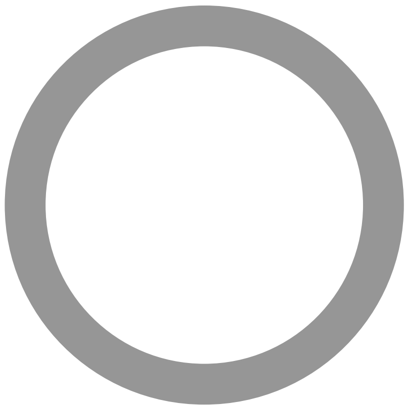
1.4. Results

During our assessment on the scoped Solera contracts, we discovered three findings, all of which were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Solera in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	0
🟩 Low	0
⬜ Informational	3



2. Introduction

2.1. About Solera

Solera contributed the following description of Solera:

> Solera is the first fully integrated DeFi suite with a custom framework for collateralizing and borrowing against RWAs and yield bearing assets on Plume network. Solera has developed a leveraged asset looping flow for ERC-20 tokens, through borrowing on Solera Lending Markets and minting/swapping into new collateral assets for leveraged yield exposure

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.

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:

Solera Contracts

Type	Solidity
Platform	EVM-compatible

Target	solera-contracts
Repository	https://github.com/SoleraMarkets/solera-contracts ↗
Version	`7f2d916fc97e6af70e8da72dbd66097896035a4e`
Programs	`looping/Looping.sol` `helpers/WrappedTokenGatewayV3.sol`

2.4. Project Overview

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

Aaron Esau
Engineer
aaron@zellic.io

Seunghyeon Kim
Engineer
seunghyeon@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 13, 2025	Kick-off call
March 13, 2025	Start of primary review period
March 17, 2025	End of primary review period

3. Detailed Findings

3.1. Minimum Nest vault's nRWA mint may restrict user

Target	Looping		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The code deposits the asset using the following function:

```
currentAmount = IAggregateToken(nRWA).deposit(borrowAmount, address(this),
    address(this));
```

```
function deposit(uint256 assets, address receiver, address controller)
    public override returns (uint256 shares) {
    if (receiver != msg.sender) {
        revert InvalidReceiver();
    }
    if (controller != msg.sender) {
        revert InvalidController();
    }

    shares =
        ITeller(address(this)).deposit(
            IERC20(asset()),
            assets,
            assets.mulDivDown(minimumMintPercentage, 10_000));

    return shares;
}
```

The highlighted value is the minimum number of shares the contract may receive after depositing the PUSD assets without reverting.

Impact

Note that the `minimumMintPercentage` may prevent the Looping contract from operating normally if the ratio of assets to shares is unexpectedly high.

Recommendations

Directly call the `deposit(address,uint256,uint256)` function to override the value.

Remediation

Note: This issue was independently discovered and fixed by Solera during the assessment.

This issue has been acknowledged by Solera, and a fix was implemented in commit `cb00bd79` ↗.

3.2. Lack of support for users who select an E-mode

Target	Looping		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

As described by Aave, High Efficiency Mode (E-mode) in v3

> ...allows users to maximize their borrowing power by selecting optimized asset categories. With the recent addition of liquid E-modes, assets may belong to multiple E-mode categories, providing greater flexibility in choosing the most efficient borrowing strategy based on category-specific parameters and asset availability.

The `_calculateBorrowAmount` and `_calculateBorrowAmountSingleAsset` functions simply select the liquidation threshold from the configuration, regardless of the value selected by the user.

Impact

Users may be unexpectedly unable to borrow, if they have set an E-mode, because they are unable to open an unhealthy position. Or otherwise, the borrow amount may not be optimal.

Recommendations

Check the user's E-mode selection with Aave V3 before defaulting to the configured liquidation threshold.

Remediation

Note: This issue was independently discovered and fixed by Solera during the assessment.

This issue has been acknowledged by Solera, and a fix was implemented in commit `c11d94d3` ↗.

3.3. Unnecessary approvals of zero tokens

Target	Looping, WrappedTokenGatewayV3		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In multiple places, the contracts first approve zero amount of a token and then approve the intended amount:

```
IERC20(params.borrowToken).approve(address(swapRouter), 0);
IERC20(params.borrowToken).approve(address(swapRouter), borrowAmount);
```

This is unnecessary, as the second approval would overwrite the first one — for example, in OpenZeppelin's ERC20 contract ↗:

```
function _approve(address owner, address spender, uint256 value,
    bool emitEvent) internal virtual {
    if (owner == address(0)) {
        revert ERC20InvalidApprover(address(0));
    }
    if (spender == address(0)) {
        revert ERC20InvalidSpender(address(0));
    }
    _allowances[owner][spender] = value;
    if (emitEvent) {
        emit Approval(owner, spender, value);
    }
}
```

Impact

Many of the functions in the two contracts will make one or more unnecessary external calls to approve zero tokens, which will increase the gas cost of these functions.

Recommendations

Remove the first approval of zero tokens, as it is unnecessary.

Remediation

Solera noted that USDT has the following requirement ↗:

```
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32)
    {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
}
```

However, we were informed that only the following tokens would be used: WPLUME, SPLUME, WETH, PETH, Nest assets (NRWA, NYIELD, NTBILL), USDC, and PUSD.

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. Test suite

When building a complex contract ecosystem 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.

The test coverage for this project should include all contracts, not just surface-level functions. It is important to test the invariants required for ensuring security and also verify mathematical properties.

Therefore, we recommend building a rigorous test suite that includes all contracts 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 may seem contradictory given the time investment to create and maintain tests. To expand upon it, 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.

In our opinion, the tests for the assessed code are inadequate for the following reasons:

- Several functions are not tested at all. While many of the functions are simple or copy significant amounts of logic from other functions, any change in logic is inherently risky and should be properly tested.
  - `Looping.loopPUSD`
  - `Looping.loopNRWA`
  - `WrappedTokenGatewayV3.loopEntryPLUMESingleSwap`
  - `WrappedTokenGatewayV3.loopEntryPLUMEMultiSwap`

    - `WrappedTokenGatewayV3.loopEntryPLUMESingleAsset`
    - `WrappedTokenGatewayV3.loopExitPLUMESingleSwap`
    - `WrappedTokenGatewayV3.loopExitPLUMEMultiSwap`
    - `WrappedTokenGatewayV3.loopExitPLUMESingleAsset`
    - `WrappedTokenGatewayV3.loopPLUMESingleAsset`
- Looping tests do not test against many positive scenarios — for example, using varying borrow amounts (prices, supply, etc.), testing with tokens of varying decimals, and so on.
- Few looping tests assert that the state was properly changed after executing functions.
- There are few tests for edge-case scenarios — for example, extreme low-liquidity situations in swap paths, extreme token decimals, and outdated price oracles (all with a range of target health factors).

  Solera added the following note regarding this point:

  > The primary / sole path for users to enter loops is through direct minting (nest, pETH, solera staking) and swap routes are not a priority / primary route.

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Plume network.

During our assessment on the scoped Solera contracts, we discovered three findings, all of which were 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.