Zellic

**Prepared for**
**Mithril Labs Ltd.**
SatLayer

**Prepared by**
**Katerina Belotskaia**
**Vlad Toie**
Zellic

**January 16, 2025**

# Deposit Contract
## Smart Contract Patch Review

## 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 SatLayer from January 14th to January 15th, 2025. During this engagement, Zellic reviewed the Deposit contract's code for security vulnerabilities, design issues, and general weaknesses in security posture.

We were asked to review a patch to the Deposit contract, which introduced a new version of the SatlayerPool contract, based on the previous version.

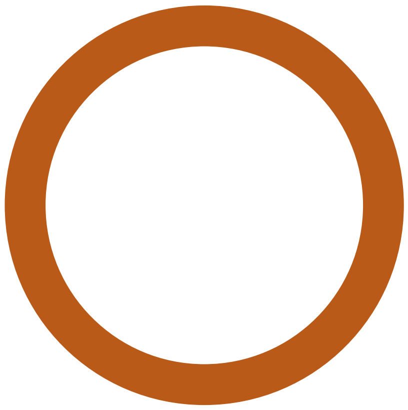
## 1.2. Results

During our assessment on the scoped Deposit contract, we discovered two findings, both of which were high impact.

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

### Breakdown of Finding Impacts

| Impact Level | Count |
|---|---|
| 🟥 Critical | 0 |
| 🟧 High | 2 |
| 🟨 Medium | 0 |
| 🟩 Low | 0 |
| ⬜ Informational | 0 |



# 2. Introduction

## 2.1. About Deposit Contract

SatLayer contributed the following description of the SatLayer project:

> SatLayer is a shared security platform designed to leverage Bitcoin as the primary security collateral. By deploying as a set of smart contracts on Babylon, SatLayer enables BTC restakers to secure any type of dApp or protocol as a Bitcoin Validated Service (BVS).

## 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:

### Deposit Contract

| | |
|---|---|
| **Type** | Solidity |
| **Platform** | EVM-compatible |
| **Target** | Only changes between 74703945...144247b9 |
| **Repository** | https://github.com/satlayer/deposit-contract ↗ |
| **Version** | `74703945...144247b9` |
| **Programs** | `ReceiptToken.sol`<br>`SatlayerPool.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 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:

**Katerina Belotskaia**
Engineer
kate@zellic.io ↗

**Vlad Toie**
Engineer
vlad@zellic.io ↗

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **January 14, 2025** | Start of primary review period |
| **January 15, 2025** | Kick-off call |
| **January 16, 2025** | End of primary review period |

# 3. Detailed Findings

## 3.1. ERC20Plugins token can lead to reentrancy issues

| | | | |
|---|---|---|---|
| **Target** | SatlayerPoolV2 | | |
| **Category** | Coding Mistakes | **Severity** | High |
| **Likelihood** | Low | **Impact** | High |

### Description

The `SatlayerPoolV2::withdraw` function does not follow the checks-effects-interactions pattern, and it contains a potential reentrancy issue.

```
function withdraw(address _token) external {
    if (withdrawAmounts[_token][msg.sender] == 0)
    revert WithdrawAmountCannotBeZero();
    if (block.timestamp < withdrawTimes[_token][msg.sender])
    revert WithdrawAttemptTooEarly();
    if (withdrawTimes[_token][msg.sender] == 0) revert WithdrawTimeNotSet();

    IERC20Metadata(_token).safeTransfer(msg.sender,
    withdrawAmounts[_token][msg.sender]);

    withdrawAmounts[_token][msg.sender] = 0;
    withdrawTimes[_token][msg.sender] = 0;
}
```

The `withdrawAmounts` and `withdrawTimes` mappings for the corresponding `token` and `msg.sender` address are updated only after the `IERC20Metadata(_token).safeTransfer` external call.

### Impact

The issue can lead to total training of the pool, should an ERC20Plugins compatbile token be used.

Assuming only legitimate ERC-20 assets are used, and therefore that the attacker cannot gain control during the call to `IERC20Metadata(_token).safeTransfer`, this issue does not constitute an exploitable vulnerability. As such, the likelihood is low.

However, careful vetting of the allowed ERC-20 assets is required unless a code change is made to mitigate the issue.

### Recommendations

We recommend strictly following the checks-effects-interactions pattern as well as updating `withdrawAmounts` and `withdrawTimes` before calling `IERC20Metadata(_token).safeTransfer`.

```
function withdraw(address _token) external {
    if (withdrawAmounts[_token][msg.sender] == 0)
    revert WithdrawAmountCannotBeZero();
    if (block.timestamp < withdrawTimes[_token][msg.sender])
    revert WithdrawAttemptTooEarly();
    if (withdrawTimes[_token][msg.sender] == 0) revert WithdrawTimeNotSet();

   IERC20Metadata(_token).safeTransfer(msg.sender, withdrawAmounts[_token][
       msg.sender]);

    withdrawAmounts[_token][msg.sender] = 0;
    withdrawTimes[_token][msg.sender] = 0;

   IERC20Metadata(_token).safeTransfer(msg.sender, withdrawAmounts[_token][
       msg.sender]);
}
```

Additionally, we recommend clearly documenting the types of ERC-20 assets that are allowed to be used with the SatlayerPoolV2 contract.

### Remediation

This issue has been acknowledged by SatLayer, and a fix was implemented in commit `d1b3111d` ↗. In addition, the SatLayer team has stated that they do not plan to integrate such tokens, and they will be mindful of this issue in the future.

## 3.2. Tokens with callback support can inflate the `actualAmount`

| | | | |
|---|---|---|---|
| **Target** | SatlayerPoolV2 | | |
| **Category** | Coding Mistakes | **Severity** | High |
| **Likelihood** | Low | **Impact** | High |

### Description

The `depositFor` function uses before- and after-balance checks during token transfers to account for the actual amount of transferred tokens.

```
function depositFor(address _token, address _for, uint256 _amount)
    whenNotPaused external {
    if (_amount == 0) revert DepositAmountCannotBeZero();
    if (_for == address(0)) revert CannotDepositForZeroAddress();
    if (!tokenAllowlist[_token]) revert TokenNotAllowedForStaking();
    if (capsEnabled && caps[_token] < getTokenTotalStaked(_token) + _amount)
    revert CapReached();

    uint256 balanceBefore = IERC20Metadata(_token).balanceOf(address(this));
    IERC20Metadata(_token).safeTransferFrom(msg.sender, address(this),
    _amount);
    uint256 actualAmount = IERC20Metadata(_token).balanceOf(address(this))
    - balanceBefore;

    emit Deposit(++eventId, _for, _token, actualAmount);

    ReceiptToken(tokenMap[_token]).mint(_for, actualAmount);
}
```

However, tokens that support callbacks during transfer (such as ERC-777) can reenter the `depositFor` function, compromising ulterior calculations.

For example, a user transfers 100 tokens during a `depositFor` call. The initial value of `balanceBefore` is 0. If a callback function is invoked in the `token` contract and the user reenters the `depositFor()` function, the new `balanceBefore` would be equal to 100 tokens. During the reentered call, another 100 tokens would be transferred, resulting in the `actualAmount` being calculated as 200 minus `balanceBefore` (which is 100), giving an `actualAmount` of 100.

Meanwhile, in the initial call, the `actualAmount` would be calculated as 200 minus `balanceBefore` (which is 0), resulting in an `actualAmount` of 200. This creates a situation where the total accounted token amount is 300, even though only 200 tokens have actually been transferred.

### Impact

This issue can lead to an inflated `actualAmount` value, which impairs the correct accounting of the total staked tokens.

### Recommendations

We recommend implementing a reentrancy-protection mechanism to prevent such attacks, especially if tokens with callback support are allowed to be used with the SatlayerPoolV2 contract.

### Remediation

This issue has been acknowledged by SatLayer, and a fix was implemented in commit `07f66277` ↗. In addition, the SatLayer team has stated that they do not plan to integrate such tokens, and they will be mindful of this issue in the future.

## 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. Patch review

The purpose of this section is to document the exact diffs of the codebase that were considered in scope for this audit.

As requested by SatLayer, we focused on the changes made between commit `74703945` ↗ and the latest commit as of the time of writing at `144247b9` ↗.

#### The SatlayerPool contract has been updated to version SatlayerPoolV2

The following changes were made in the update.

- The contract now supports upgradability using the transparent proxy pattern.
- A withdraw timeout period has been introduced: `WITHDRAW_PERIOD = 7 days`.
- The withdrawal process has been updated to a two-step process:
    1. The new `queueWithdrawal` function must be called by the user to initiate the withdrawal of tokens. The `withdrawTimes` for `msg.sender` will be set to `block.timestamp + WITHDRAW_PERIOD`. The `withdrawAmounts` for `msg.sender` will be increased by the desired withdrawal amount. The corresponding amount of liquidity-pool tokens will be burned.
    2. After the `WITHDRAW_PERIOD` expires, the `withdraw` function can be executed by the `msg.sender` who initiated the withdrawal. The queued amount of tokens will be transferred to the caller. The `withdrawAmounts[_token][msg.sender]` and `withdrawTimes[_token][msg.sender]` will be reset.
- The `setCapsEnabled` function now reverts if the same value is being set again.
- The `setTokenStakingParams` function has been updated to allow changes to the `_canStake` and `_cap` parameters only if the new values differ from the current ones.
- The `migrate`, `setCap`, and `setMigrator` functions have been removed.

## 5. Assessment Results

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

During our assessment on the scoped Deposit contract, we discovered two findings, both of which were high impact.

### 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.