Zellic

# SPL Single Stake Pool

Smart Contract Security Assessment

Jun 21, 2023

*Prepared for:*

Solana Foundation

*Prepared by:*

**Jasraj Bedi and Filippo Cremonese**

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 Solana Foundation from May 11th to May 23rd, 2023. During this engagement, Zellic reviewed SPL Single Stake Pool'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 users able to withdraw more than they are entitled?
- Can deposits or withdrawals be manipulated to unfairly reward or penalize users?
- Is it possible for any role to lock user funds in the contract?

## 1.2 Non-goals and Limitations

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

- Problems relating to the front-end components and infrastructure of the project
- Problems due to improper key custody or off-chain access control
- Issues stemming from code or infrastructure outside of the assessment scope

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

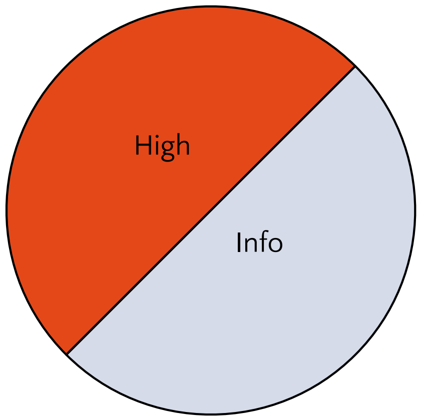
## 1.3 Results

During our assessment on the scoped SPL Single Stake Pool contracts, we discovered two findings. One relatively trivial high severity issue was found in addition to a second issue of informational nature.

Additionally, Zellic recorded its notes and observations from the assessment for Solana Foundation's benefit in the Discussion section (4) at the end of the document.

## Breakdown of Finding Impacts

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



# 2 Introduction

## 2.1 About SPL Single Stake Pool

SPL Single Stake Pool is a stripped-down stake pool program for the single-validator case that enables liquid staking with no fees and 100% capital efficiency.

## 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 the contracts' 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 code base 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.

## 2.3 Scope

The engagement involved a review of the following targets:

### SPL Single Stake Pool Programs

| | |
|---|---|
| **Repository** | https://github.com/solana-labs/solana-program-library |
| **Version** | solana-program-library: `9dbdc3bdae31dda1dcb35346aab2d879deecf194` |
| **Program** | • SPL Single Stake Pool |
| **Type** | Rust |
| **Platform** | Solana |

## 2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of three person-weeks. The assessment was conducted over the course of two calendar weeks.

### 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:

**Jasraj Bedi**, Co-founder
jazzy@zellic.io

**Filippo Cremonese**, Engineer
fcremo@zellic.io

## 2.5 Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **May 11, 2023** | Start of primary review period |
| **May 23, 2023** | End of primary review period |

# 3 Detailed Findings

## 3.1 Incorrect vote account deserialization

- **Target**: `UpdateTokenMetadata`
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

### Description

The `process_update_pool_token_metadata` function, which processes the `UpdateTokenMetadata` instruction, manually deserializes parts of the vote account to ensure the transaction has been signed by the withdraw authority of the vote account.

```
fn process_update_pool_token_metadata(/* [...] */) -> ProgramResult {
    // [...]
    // we use authorized_withdrawer to authenticate the caller controls
    the vote account
    // this is safer than using an authorized_voter since those keys live
    hot
    // and validator-operators we spoke with indicated this would be their
    preference as well
    let vote_account_data = &vote_account_info.try_borrow_data()?;
    let vote_account_withdrawer = vote_account_data
        .get(VOTE_STATE_START..VOTE_STATE_END)
        .map(Pubkey::new)
        .ok_or(SinglePoolError::UnparseableVoteAccount)?;

    if *authorized_withdrawer_info.key != vote_account_withdrawer {
        msg!("Vote account authorized withdrawer does not match the
    account provided.");
        return Err(SinglePoolError::InvalidMetadataSigner.into());
    }
    if !authorized_withdrawer_info.is_signer {
        msg!("Vote account authorized withdrawer did not sign metadata
    update.");
        return Err(SinglePoolError::SignatureMissing.into());
    }
```

However, the `VOTE_STATE_START` and `VOTE_STATE_END` constants are incorrectly assigned the values of `4` and `36`, respectively; this means the `node_pubkey` field of the vote account is read instead of the intended `authorized_withdrawer`:

```
pub struct VoteState {
    /// the node that votes in this account
    pub node_pubkey: Pubkey,
    /// the signer for withdrawals
    pub authorized_withdrawer: Pubkey,

    // [...]
}
```

Note that vote accounts are actually a serialized instance of the `VoteStateVersions` type; therefore, an additional four bytes must be added for the enum discriminant:

```
pub enum VoteStateVersions {
    V0_23_5(Box<VoteState0_23_5>),
    Current(Box<VoteState>),
}
```

## Impact

The incorrect account was allowed to update the MPL token metadata.

## Recommendations

The easiest and quickest remediation would be to set `VOTE_STATE_START = 36` and `VOTE_STATE_END = 68`.

However, we also encourage the development team to consider using functions provided by the official SDK, such as `VoteState::from` or `VoteState::deserialize`.

## Remediation

This issue has been acknowledged by Solana Foundation, and a fix was implemented in commit 34ff3aba.

## 3.2 Missing authority account check

- **Target**: `DepositStake`
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Informational

### Description

Stake accounts have two authorities: a withdrawer and a staker. Before invoking `DepositStake` to deposit a stake account into a pool, the caller must transfer the authorities of the stake account to the stake pool program.

However, the `process_deposit_stake` function, which processes `DepositStake` instructions, does not check whether the withdraw authority of the stake account being deposited has been transferred correctly.

### Impact

The withdraw authority of the stake account being merged can still be assigned to a third party after it has been deposited. Fortunately, depositing a stake account is implemented by merging it into the pool stake account (controlled entirely by the pool authority), draining the input stake account of all its lamports; therefore, it is not possible to withdraw lamports that have been deposited.

As the issue is not exploitable for profit, it is reported as informational.

### Recommendations

We recommend to require both the withdraw and stake authority to be transferred to the pool program to accept a deposit. This serves as a defense to guard against possible exploits that might be enabled by a future iteration of the program.

### Remediation

This issue has been acknowledged by Solana Foundation.

The withdraw authority is implicitly validated by the stake system program when the source stake account is merged into the pool stake account, as the authorities for both accounts must match exactly. This check is performed by the `metas_can_merge` function.

# 4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

## 4.1 Test suite

The repository lacks a comprehensive test suite for the program being audited at the commit we reviewed. We note that a test suite is under development at the time of writing under the namespace of an individual member of the Solana development team. In our opinion, the test suite in its current version could be expanded to test the entire functionality of the contract, as currently the individual handlers of the program instructions are not being tested directly.

## 4.2 Authorization scheme

The authorization scheme used by the program is quite simple — a single PDA is derived from the address of every vote account. This PDA is used for multiple purposes, including as mint authority, as withdrawer and staker authority for the pool stake account, and as authority for the MPL registry.

On one hand, this simple authorization scheme is easy to understand and reason about; on the other, reusing the same PDA implies that a compromise to any external program receiving the authority PDA as a signer would also lead to a compromise of the assets/data controlled by the same authority.

Note: this concern was addressed in commit 00cba617 as part of a larger pull request. The commit defines three PDAs that have separate authorities, respectively for minting, staking, and managing MPL metadata.

# 5 Threat Model

This provides a full threat model description for the various instructions supported by the program, which analyzes common threats faced by Solana programs. In addition, some potential threats are not discussed individually, and are documented in the following paragraphs.

**Reentrancy**: Generally, reentrancy is not a concern for Solana programs, as only self-reentrancy is permitted by the runtime. The program does not use self-reentrancy, nor is there any possibility to force an unintended self-reentrant call.

**Malicious external calls / confused deputy**: All the external program accounts are checked and all calls are made to the intended program; moreover, almost all external calls are made to standard built-in programs or SPL programs, with the exception of the MPL registry.

**Account type confusion**: The stake pool does not define any new account type. As such, the responsibility to check for account type confusion is largely delegated to external programs.

## 5.1 Legend

The following shorthand is used to denote particular properties about the Solana accounts.

| Key | Property |
|---|---|
| w | Writable |
| s | Signer |

## 5.2 Instructions

### `InitializePool`

This instruction can be used to initialize accounts required by the stake pool.

#### Accounts

- `[]` Validator vote account

  - ☑ Addr checked? Not needed.
  - ☑ Owner checked? Must be vote program.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[w]` Pool stake account
  - ☑ Addr checked? PDA derived from the vote account address.
  - ☑ Owner checked? Is written to (so it is owned by the pool program; it is transferred to the stake program).
  - ☑ Rent issues? Checked.
  - ☑ Account type confusion? N/A.
- `[]` Pool authority
  - ☑ Address checked? Unique PDA derived from the vote account address.
  - ☑ Owner checked? N/A.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[w]` Pool token mint
  - ☑ Address checked? Unique PDA derived from the vote account address.
  - ☑ Owner checked? Is allocated by the instruction and assigned to the token program.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` Rent sysvar
  - ☑ Address checked? Checked.
  - ☑ Owner checked? N/A.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` Clock sysvar
  - ☑ Address checked? N/A (not used by this program, just passed in CPIs).
  - ☑ Owner checked? N/A.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` Stake history sysvar
  - ☑ Address checked? N/A (not used by this program, just passed in CPIs).
  - ☑ Owner checked? N/A.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` Stake config sysvar
  - ☑ Address checked? N/A (not used by this program, just passed in CPIs).

  - ☑ Owner checked? N/A.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` System program
  - ☑ Address checked? Checked.
  - ☑ Owner checked? N/A.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` Token program
  - ☑ Address checked? Checked.
  - ☑ Owner checked? N/A.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` Stake program
  - ☑ Address checked? Has to be the stake program.
  - ☑ Owner checked? N/A.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.

### `DepositStake`

This instruction can be used to deposit a stake account into the pool, receiving pool tokens in exchange. The amount of tokens minted to the user is equal to `added_stake * supply / total_pool_assets_before_deposit`.

#### Accounts

- `[w]` Pool stake account
  - ☑ Address checked? PDA derived from the vote account address — must differ from the user stake account.
  - ☑ Owner checked? Not needed (account is written).
  - ☑ Rent issues? Not needed.
  - ☑ Account type confusion? First four bytes are used as discriminator.
- `[]` Pool authority
  - ☑ Address checked? PDA derived from the vote account address.
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.

- `[w]` Pool token mint
  - ☑ Address checked? PDA derived from the vote account address
  - ☑ Owner checked? Not needed (account is written)
  - ☑ Rent issues? NA
  - ☑ Account type confusion? NA
- `[w]` User stake account to join to the pool
  - ☑ Address checked? Must differ from the pool stake account, otherwise no further checks are needed.
  - ☑ Owner checked? Not needed (account is written).
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[w]` User account to receive pool tokens
  - ☑ Address checked? Not needed.
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? Not needed.
  - ☑ Account type confusion? N/A.
- `[w]` User account to receive lamports
  - ☑ Address checked? Not needed.
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? Not needed.
  - ☑ Account type confusion? N/A.
- `[]` Clock sysvar
  - ☑ Address checked? Yes.
  - ☑ Owner checked? N/A.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` Stake history sysvar
  - ☑ Address checked? N/A (not used by this program, just passed in CPIs).
  - ☑ Owner checked? N/A.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` Token program
  - ☑ Address checked? Checked.
  - ☑ Owner checked? N/A.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` Stake program
  - ☑ Address checked? Checked.

- ☑ Owner checked? N/A.
- ☑ Rent issues? N/A.
- ☑ Account type confusion? N/A.

### `WithdrawStake`

This instruction allows to redeem pool tokens in exchange for a stake account with the corresponding amount of staked lamports. Before calling this instruction, the pool program must be authorized to withdraw the pool tokens from the withdrawer account.

#### Accounts

- `[w]` Pool stake account
  - ☑ Addr checked? PDA derived from the vote account address — different from the user destination stake account.
  - ☑ Owner checked? Not needed (account is written).
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` Pool authority
  - ☑ Addr checked? PDA derived from the vote account address.
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[w]` Pool token mint
  - ☑ Addr checked? PDA derived from the vote account address.
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[w]` User stake account to receive stake at
  - ☑ Addr checked? No check needed (apart from being different to the pool stake account).
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[w]` User account to take pool tokens from
  - ☑ Addr checked? Not needed.
  - ☑ Owner checked? Not needed (account is passed to the token program that verifies it owns it and also writes to it).

  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` Clock sysvar
  - ☑ Addr checked? Not needed (not used directly).
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` Token program
  - ☑ Addr checked? Checked to be spl-token.
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` Stake program
  - ☑ Addr checked? Checked to be the system stake program.
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.

### `CreateTokenMetadata`

This instruction can be used to initialize the MPL token metadata for the tokens representing pool deposits.

#### Accounts

- `[]` Pool authority
  - ☑ Addr checked? PDA derived from the vote account address.
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` Pool token mint
  - ☑ Addr checked? PDA derived from the vote account address.
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[s, w]` Payer for creation of token metadata account
  - ☑ Addr checked? Not needed (just pays for the transaction).
  - ☑ Owner checked? Has to be the system account.

- ☑ Rent issues? N/A.
- ☑ Account type confusion? N/A.

- `[w]` Token metadata account
  - ☑ Addr checked? Checked to be the correct PDA.
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? Not needed (MPL responsibility).
  - ☑ Account type confusion? N/A.
- `[]` Metadata program ID
  - ☑ Addr checked? Checked to be the official MPL program.
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[]` System program ID
  - ☑ Addr checked? Checked to be the system program.
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.

### `UpdateTokenMetadata`

This instruction can be used to update the MPL metadata of the token representing pool deposits.

#### Accounts

- `[]` Validator vote account
  - ☑ Addr checked? Not needed (but other PDAs are derived from this account address).
  - ☑ Owner checked? Yes, must be vote program.
  - ☑ Rent issues? N/A.
  - ☐ Account type confusion? Checked (was incorrect, see Finding 2).
- `[]` Pool authority
  - ☑ Addr checked? PDA derived from the vote account.
  - ☑ Owner checked? Not needed.
  - ☑ Rent issues? N/A.
  - ☑ Account type confusion? N/A.
- `[s]` Vote account authorized withdrawer
  - ☑ Addr checked? Must be the authorized withdrawer of the vote account.

    - ☑ Owner checked? N/A.
    - ☑ Rent issues? N/A.
    - ☑ Account type confusion? N/A.
- `[w]` Token metadata account
    - ☑ Addr checked? PDA derived from the mint account address.
    - ☑ Owner checked? Not needed (MPL responsibility, and it gets written to).
    - ☑ Rent issues? N/A.
    - ☑ Account type confusion? Not needed (MPL responsibility).
- `[]` Metadata program ID
    - ☑ Addr checked? Checked to be the correct MPL program.
    - ☑ Owner checked? N/A.
    - ☑ Rent issues? N/A.
    - ☑ Account type confusion? N/A.

# 6 Audit Results

At the time of our audit, the audited code was not deployed to mainnet Solana.

During our assessment on the scoped SPL Single Stake Pool contracts, we discovered two findings. No critical issues were found. One was of high impact and the remaining finding was informational in nature.

## 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 audit version of 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.

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.