Solana Stake Pool

Audit

Presented by:

OtterSec contact@osec.io

Filippo Barsanti barsa@osec.io

Harrison Green hgarrereyn@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Solana engaged OtterSec to perform an assessment of the `Solana Stake Pool` program. This assessment was conducted between January 2nd and January 20th, 2023. For more information on our auditing methodology, see Appendix B.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team to streamline patches and confirm remediation. We delivered final confirmation of the patches January 20th, 2023.

Key Findings

Over the course of this audit engagement, we produced 6 findings total.

In particular, we identified two potential vulnerabilities that could allow a malicious pool operator to steal user funds via front-running a deposit (OS-SSP-ADV-00) and bypassing the fee update delay (OS-SSP-ADV-01).

We also made several recommendations around logical inconsistencies such as on deposit/withdraw fees (OS-SSP-SUG-02) and transient stake accounts (OS-SSP-SUG-03). These issues may lead to unexpected behavior but do not present immediate security concerns.

Overall, we commend the Solana team for being responsive and knowledgeable throughout the audit.

02 | Scope

The source code was delivered to us in a git repository at github.com/solana-labs/solana-program-library/tree/master/stake-pool. This audit was performed against commit `eba709b`.

A brief description of the programs is as follows.

The Solana stake pool provides a way for users to stake SOL in an automated way, without having to worry about choosing the right validator and maximizing the rewards. A stake pool can be automated through an off chain program that fetches information about validators and the blockchain and decides whether to rebalance the stakes of each validator. These rebalance operations can also be executed manually.

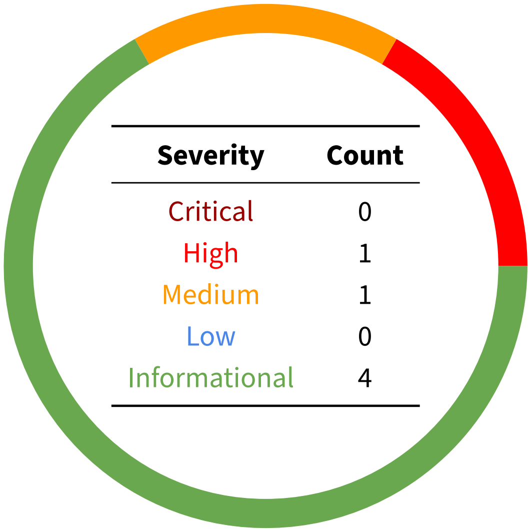
In exchange for their deposit (SOL or stake), the user receives SPL tokens representing their fractional ownership in pool, which can then be later exchanged for SOL or an active stake account through a withdraw operation when a user wishes to exit the pool.

Rewards for staked tokens are based on the current inflation rate, total number of SOL staked on the network, and an individual validator's uptime and commission (fee).

03 | Findings

Overall, we report 6 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	1
Medium	1
Low	0
Informational	4

04 | Vulnerabilities

Here we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-SSP-ADV-00	High	Resolved	Pool manager can front-run deposits to steal pool tokens.
OS-SSP-ADV-01	Medium	Resolved	Fee update delay can be bypassed at epoch boundary.

OS-SSP-ADV-00 [high] [resolved] | Front-Run Deposits To Steal Pool Tokens

Description

There is no delay mechanism on updating deposit fees and an upper bound of 100%. A user that submits a large deposit could be front-run by a pool manager who sets the fee close to 100% (ensuring that user still receives 1 pool token), receives almost the full value of the deposit, and then reduces the fee afterward.

Remediation

Add instructions that allow the user to specify the minimum amount of tokens they expect to receive.

Patch

Fixed in #3980 with the introduction slippage instruction variants.

In order to protect against manager fee hikes, a user can use the following instructions and specify a `slippage` parameter. If the subsequent fee deduction reduces the received amount below this value, the instruction will abort:

1. `DepositStakeWithSlippage`
2. `WithdrawStakeWithSlippage`
3. `DepositSolWithSlippage`
4. `WithdrawSolWithSlippage`

OS-SSP-ADV-01 [med] [resolved] | Fee Update Delay Bypass

Description

Certain fees (StakeWithdrawal, SolWithdrawal, Epoch) can only be updated in the next epoch, as show in the snippets below.

program/src/processor.rs RUST

```rust
if fee.can_only_change_next_epoch() && stake_pool.last_update_epoch <
    ↪  clock.epoch {
    return Err(StakePoolError::StakeListAndPoolOutOfDate.into());
}
```

program/src/state.rs RUST

```rust
#[inline]
pub fn can_only_change_next_epoch(&self) -> bool {
    matches!(
        self,
        Self::StakeWithdrawal(_) | Self::SolWithdrawal(_) |
    ↪   Self::Epoch(_)
    )
}
```

In the current implementation, the manager can update the fee at the end of the current epoch and then immediately apply the fee at the start of the next epoch.

For example, a pool manager could use this technique to instantaneously apply a withdrawal fee of 0.1% (the baseline fee).

Remediation

To give users sufficient time to adapt to changing fees, it would likely make sense to ensure the proposed fee is blocked for at least one full epoch.

Patch

Fixed in #3979. Fees now require at least one full epoch before they can be applied.

05 | General Findings

Here we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-SSP-SUG-00	Rent-exempt calculations are performed on theoretical data size rather than actual.
OS-SSP-SUG-01	Setting fees to 100% leads to deposit and withdraw operations being disabled.
OS-SSP-SUG-02	Deactivating a validator requires reducing its active stake to the minimum delegation.
OS-SSP-SUG-03	Transient stake accounts can be reused but can only move in one direction.

OS-SSP-SUG-00 | Rent-Exempt Calculated On Theoretical Data Size

Description

Ensure rent-exempt calculations are performed on actual data size instead of theoretical data size. E.g. in `AddValidatorToPool`, rent exempt calculation on the reserve_pool is performed on the default account size:

program/src/processor.rs RUST

```rust
let space = std::mem::size_of::<stake::state::StakeState>();
let stake_minimum_delegation = stake::tools::get_minimum_delegation()?;
let required_lamports = minimum_delegation(stake_minimum_delegation)
        .saturating_add(rent.minimum_balance(space));
```

If it was possible to create a `Stake` account larger than the default (200 bytes), this computation would be inaccurate and the stake account may be able to be closed here.

Note: the stake program validates the account size during initialization and during `split` the remaining lamports must be either greater than the minimum requirement or zero.

Remediation

Calculate minimum rent on the *actual* size of the account.

Patch

Fixed in #4000. The minimum rent is now calculated based on the same value used to allocate the account space, instead of always using the theoretical size.

OS-SSP-SUG-01 | Setting Fees To 100% Breaks Deposit And Withdraw

Description

Setting a deposit or withdraw fee of 100% will always break deposits and withdraws since the amount of lamports/tokens the user will receive is 0, hence failing the instruction.

The following snippet shows the check performed on the amount of lamports the user would receive, after subtracting fees, that leads to the failure of the withdraw instruction.

program/src/processor.rs RUST

```rust
let pool_tokens_fee = if stake_pool.manager_fee_account ==
    ↪  *burn_from_pool_info.key
    || stake_pool.check_manager_fee_info(manager_fee_info).is_err()
{
    0
} else {
    stake_pool
        .calc_pool_tokens_sol_withdrawal_fee(pool_tokens)
        .ok_or(StakePoolError::CalculationFailure)?
};
let pool_tokens_burnt = pool_tokens
    .checked_sub(pool_tokens_fee)
    .ok_or(StakePoolError::CalculationFailure)?;

let withdraw_lamports = stake_pool
    .calc_lamports_withdraw_amount(pool_tokens_burnt)
    .ok_or(StakePoolError::CalculationFailure)?;

if withdraw_lamports == 0 {
    return Err(StakePoolError::WithdrawalTooSmall.into());
}
```

Remediation

Consider disallowing fees to be equal to 100%.

OS-SSP-SUG-02 | Deactivating Validator With Less Than 2x Min Delegation

Description

Deactivating a validator requires reducing its active stake to the minimum delegation first. However, increase and `DecreaseValidatorStake` instructions require a minimum of the same `minimum_stake` amount. So, for example, a validator at 1.9x min delegation would require the following sequence to deactivate:

1. `IncreaseValidatorStake` - bring amount to 2.9x min delegation
2. `DecreaseValidatorStake` - reduce amount to 1.0x min delegation
3. `RemoveValidatorFromPool`

Remediation

For ease of validator removal, consider allowing a validator to be removed if they have <= 2x minimum delegation instead of the current <= 1x minimum delegation.

Patch

Fixed in #3999. The lamport check was completely removed.

OS-SSP-SUG-03 | Transient Stake Account Only Moves In One Direction

Description

A transient stake account represents stake that is either activating or deactivating.

The `IncreaseValidatorStake` and `DecreaseValidatorStake` instructions ensure that only one transient stake account is active at a time. However, the newly added `*AdditionalValidatoStake` and `Redelegate` instructions can make use of existing transient stake accounts.

In the case where multiple `(Increase/Decrease)ValidatorStake` instructions are issued in the same epoch that the validator was added to these pool, the ephemeral accounts can merge with the transient account and the resulting stake will move in only one direction. Either it will deactivate and merge with the reserve pool if the first instruction was `DecreaseValidatorStake` or it will activate and merge with the validator stake if the first instruction was `IncreaseValidatorStake`.

For example, the sequence:

1. `AddValidatorToPool`
2. `IncreaseValidatorStake(A)`
3. `DecreaseAdditionalValidatorStake(B)`

will result in *only* a net transfer of A lamports to the validator (not `(A-B)` as might be expected).

Note that this is only possible in the first epoch that the validator is added to the pool. In subsequent epochs, the ephemeral stake will be unable to merge with the transient stake when both increase and decrease instructions are issued.

Remediation

Consider setting a direction flag in the validator's info when a stake transfer begins. Prohibit transfers in the opposite direction until the transient stake has been resolved.

Patch

Fixed in #3987. An explicit check to disallow DecreaseAdditionalValidatorStake operations if the transient account is activating/active, and IncreaseAdditionalValidatorStake and Redelegate operations if the transient account is deactivating/inactive was added.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical

Vulnerabilities that immediately lead to loss of user funds with minimal preconditions

Examples:

- Misconfigured authority or access control validation
- Improperly designed economic incentives leading to loss of funds

High

Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions
- Exploitation involving high capital requirement with respect to payout

Medium

Vulnerabilities that could lead to denial of service scenarios or degraded usability.

Examples:

- Malicious input that causes computational limit exhaustion
- Forced exceptions in normal user flow

Low

Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions

Informational

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants
- Improved input validation

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.