Security Audit – Lido on Solana

Draft v0.5

Neodyme

Technical-Lead: Thomas Lambertz

2021-08-31

Contents

Project Summary 3

Introduction to Solido 3

Methodology 4

Findings 5

Attackers Can Mint Unlimited stSOL due to Missing Reserve Account Check (Critical) 6
SPL Account Closing May Cause Problems (Low) 8
RemoveValidator Instruction has Incomplete Manager Verification (Low) 9
Deletion of Rewards with Possible Economic Attack (Low) 10
Late Implementation of Unstake Accounts and Denial of Withdrawal (Low) 12
Missing Owner Checks in Account Validation (Info) 13
Missing Checks on Mint (Info) 14
Config Accounts are not Program Derived Accounts (Info) 15

Project Summary

Lido engaged Neodyme to do a detailed security analysis of their on-chain program. A thorough audit was done from August 25th to August 31st.

Main target of the audit was the source-code of the solido on-chain program in version `v0.5`, specifically commit hash `8dc658ea6913b63bc3e15e381c91c08ed61e2838`.

The code is found to be written to a high standard, and the team expertly responded to all questions.

Nevertheless, the audit revealed one major vulnerability and a few low-priority findings, which were reported and subsequently fixed by the Lido team. This report describes these findings in detail.

Introduction to Solido

Lido is a liquid staking protocol. It currently supports Ethereum and Terra, intending to extend support to the Solana blockchain. The Solana smart-contract is called `solido` and is the target of this audit.

Users can deposit the native Solana `SOL` token into the solido contract, which is then staked to a selected set of validators in a uniform distribution. The user receives `stSOL`, a token representing their share of the staked value, which can be freely traded while the staked `SOL` is accruing staking rewards. To end staking, the user can either swap their `stSol` for an active stake account or just sell the `stSol` on the open market. `stSol` automatically appreciates in value via the rewards generated by staking.

Because of the nature of the program, it is expected to hold large amounts of funds.

The solido source-code is public, and documentation which contains information for end-users, but also some internals, is available at:

- Contract: https://github.com/ChorusOne/solido
- Documentation: https://chorusone.github.io/solido

Methodology

Neodyme's audit team performed a comprehensive examination of the solido contract. The team, which consists of security engineers with extensive experience in Solana smart contract security, reviewed and tested the code of the on-chain contract, paying particular attention to the following:

- Ruling out common classes of Solana contract vulnerabilities, such as:
  - Missing ownership checks,
  - Missing signer checks,
  - Signed invocation of unverified programs,
  - Solana account confusions,
  - Re-initiation with cross-instance confusion,
  - Missing freeze authority checks,
  - Insufficient SPL-Token account verification,
  - Missing rent exemption assertion,
  - Casting truncation,
  - Arithmetic over- or underflows,
  - Numerical precision errors,
- Checking for unsafe design which might lead to common vulnerabilities being introduced in the future,
- Checking for any other, as-of-yet unknown classes of vulnerabilities arising from the structure of the Solana blockchain,
- Ensuring that the contract logic correctly implements the project specifications,
- Examining the code in detail for contract-specific low-level vulnerabilities,
- Ruling out denial of service attacks,
- Ruling out economic attacks,
- Checking for instructions that allow front-running or sandwiching attacks,
- Checking for rug-pull mechanisms or hidden backdoors.

Findings

This section discusses solido's overall design, followed by a detailed description of all our findings and their resolutions.

Every program on Solana has to store data somehow. There are multiple approaches to this, and solido chose one of the safest ones: each instance of the solido program has a single unique account that stores all data. While solido allows for multiple instances to exist simultaneously, it is written in a way that rules out many typical Solana vulnerabilities by design. By storing all state in a single account that is subsequently required in every operation, all attacks that work by confusing between multiple program-owned accounts are by definition impossible. This approach has the minor drawback that all solido contract invocations have to happen sequentially, but that should not be an issue due to Solana's performance.

The program code is well-structured and readable, contains helpful comments, and the commit messages are descriptive.

Most Solana contracts are deployed using the `upgradable loader`. An upgrade authority can, at any time, make changes to the deployed contract. As solido is not deployed to the Solana-Mainnet yet, we cannot make any assertions as to the contract-upgradability. However, unlike many other projects, Lido already has a distributed set of trusted owners, which they use in a multisig contract. This provides a higher level of protection, as the *decentralized autonomous organization* (DAO) has to commit any upgrade, ruling out a single-party rug-pull by the developer.

In total, there was one `critical`, four `low`, and three `info` findings.

Name	Severity
Attackers Can Mint Unlimited stSOL due to Missing Reserve Account Check	Critical
RemoveValidator Instruction has Incomplete Manager Verification	Low
SPL Account Closing May Cause Problems	Low
Deletion of Rewards with Possible Economic Attack	Low
Late Implementation of Unstake Accounts and Denial of Withdrawal (Low)	Low
Missing Owner Checks in Account Validation	Info
Missing Checks on Mint	Info
Config Accounts are not PDAs	Info

Attackers Can Mint Unlimited stSOL due to Missing Reserve Account Check (Critical)

Severity	Impact	Affected Component
Critical	Loss of Funds	`ProcessDeposit`

The deposit function `process_deposit` has a critical security vulnerability caused by a missing account check. This allows an attacker to mint and receive unlimited `stSOL`, which allows him to asymptotically steal all funds from the pool.

Specifically, it is not verified that the reserve account passed to `process_deposit` the correct one. Instead, the contract should validate it against `lido.reserve_account`, which is easily done with `fn check_reserve_account()` used by all other functions which access the reserve.

An attacker can thus supply an attacker-controlled reserve account that he passes to `process_deposit` instead of the actual reserve. The deposit function will transfer all deposited `SOL` into the reserve account and mint the appropriate number of `stSol` tokens. But since the attacker controls the reserve account, he can withdraw the `SOL` from there and has minted `stSol` from nothing. Since he retains ownership of his `SOL`, he can repeat this process as many times as he wants, generating an arbitrary number of `stSOL`. He can then use the `stSOL` to withdraw `SOL` from the contract, asymptotically allowing him to drain the contract of all staked `SOL`.

The following is the relevant part of the code:

```
pub fn process_deposit(
    program_id: &Pubkey,
    amount: Lamports,
    accounts_raw: &[AccountInfo],
) -> ProgramResult {
    let accounts = DepositAccountsInfo::try_from_slice(accounts_raw)?;
    if amount == Lamports(0) {                                    // ^
        msg!("Amount must be greater than zero");                 // |
        return Err(ProgramError::InvalidArgument);      //no check  |
    }                                                             // |
    let mut lido = deserialize_lido(program_id, accounts.lido)?; // v

    invoke(
        &system_instruction::transfer(accounts.user.key, accounts.
            reserve_account.key, amount.0),
        &[
            accounts.user.clone(),
            accounts.reserve_account.clone(), // SOL is transferred to
                unchecked reserve
            accounts.system_program.clone(),
        ],
```

```
20      )?;
21      /* [...] */
```

Solution The Lido team responded immediately by adding a regression test and adding the missing check. The latter was done by calling the pre-existing function `check_reserve_account`. Neodyme verified the fix.

References:

- https://github.com/ChorusOne/solido/pull/383

SPL Account Closing May Cause Problems (Low)

Severity	Impact	Affected Component
Low	Potential denial of rewards	SPL Token Interactions

The Solana token program has a `CloseAccount` instruction. It can be used by the token account owner to delete his account.

This account closing can cause issues in solido because it uses three types of externally controlled SPL-Token accounts: Validator-Rewards, Developer, and Treasury.

If the developer or treasury accounts are deleted, all invocations to `CollectValidatorFee` will fail, as it unconditionally mints `stSol` tokens as fees to the developer and treasury accounts. When these accounts cease to exist, the minting will fail. This also means solido will stop accruing staking rewards, as nobody can withdraw from the validators-vote accounts anymore.

Since the validator rewards are paid out asynchronously, there is no direct impact to the contract when a validator closes his token account. However, it will prevent said validator from being removed, as he will be stuck with an internal balance greater than zero, which nobody can withdraw. This has no benefit for the validator. The only adverse effect is that this validator will bloat the validator list indefinitely.

Solution The Lido team responded by analyzing the risk for each of the three externally controlled SPL-Token accounts. We paraphrase:

- The treasury account is controlled by the DAO. It is assumed that the DAO will act in the interest of the contract.
- The developer account is controlled by the developers, which are assumed to be trusted.
- The accounts into which validator rewards are payed are indeed controlled externally, and a malicious validator can block removal of their validator from the validator list. However, this has only cosmetic effects.

References:

- https://github.com/ChorusOne/solido/issues/400

RemoveValidator Instruction has Incomplete Manager Verification (Low)

Severity	Impact	Affected Component
Low	Required permissions for validator removal are unclear	`RemoveValidator`

The manager in the `RemoveValidator` instruction is insufficiently validated. It is only asserted that the manager has signed the transaction but never that the manager matches the account stored in `lido.manager`. This does not have any security implications yet, since validators can only be removed after they are unstaked and have no stake accounts anymore. For a previously active validator, this can only happen when the manager is in the process of removing it anyway. Furthermore, an attacker cannot prevent a new validator from being added, since adding the validator and staking to it can be wrapped into a single transaction.

Solution The Lido team removed the maintainer signature requirement as it was a leftover from a previous iteration.

References:

- https://github.com/ChorusOne/solido/pull/388

Deletion of Rewards with Possible Economic Attack (Low)

Severity	Impact	Affected Component
Low	Economic Attack, Loss of Rewards	Exchange Rate Update/Deposits

Solido's `SOL` to `stSol` exchange rate is updated once after the beginning of each epoch. This update has to be triggered. There is a short window of time in between the epoch boundary and this update, during which withdrawals are blocked, but deposits are allowed.

The fact that deposits are allowed allows an unlikely economic attack. In essence, an attacker can cause stake to deactivate for free, which will lower the rewards solido generates for all users. However, because of the nature of the attack, it is deemed very unlikely to happen.

Details When a user deposits `SOL` right after the epoch boundary, but before the exchange rate is updated, he receives an `stSOL` amount that is calculated via the old exchange rate. Note that this does not give the user a direct advantage. A user could have deposited the `SOL` exactly one slot before the epoch boundary and would have gotten the same amount of `stSol`.

But there is another relevant interaction between the solido contract and the real epoch boundary: When a user withdraws funds, solido returns an active stake account, which is subject to one epoch of deactivation time. At the real epoch boundary, funds that have been withdrawn in the previous epoch become available to the withdrawee. In particular, they are available to be re-deposited at the same exchange rate they were withdrawn at.

This does lead to an interesting, albeit slightly far-fetched attack in which an attacker can cause the contract to miss out on staking rewards. The attack works as follows:

Immediately before an epoch boundary, the attacker buys or lends a large amount of `stSOL`. He directly uses this to withdraw at the old exchange rate, receiving a delegated stake account which he immediately deactivates. After the epoch boundary, he has full control of the `SOL` in the stake account. The attacker immediately re-deposits these `SOL` into the Lido contract – before the update of the exchange rate is triggered – and receives the same number of `stSOL` as he had before he withdrew (note that there are no deposit or withdraw fees). He can now sell these tokens or repay his loan.

So far, this did not create any financial gain for the attacker. However, the contract has now missed out on the staking rewards for the attacker's share of `SOL`, since it had to pay out a delegated stake account and has received unstaked `SOL`. This causes the inherent value of the `stSOL` token to increase less as it otherwise would have, as the appreciation from the attacker's staking rewards is missing.

This under-performance would presumably affect the market price of `stSOL`, causing it to drop after the attack is executed since the assumed increase in the inherent value of `stSOL` was already priced in. Hence if the attacker shorts the price of `stSOL` before executing the attack, he may benefit.

Note that this attack requires multiple market conditions to be met, and potential gains are low. Market fees for shorting `stSOL` would probably outweigh the potential financial gain. We hence estimate the probability of the attack being executed as very low.

Solution Lido agreed that the probability of an exploit is extremely low and that the incentive for an attacker is minimal. Additionally, they argued that the risk of having to race the maintenance bot before the exchange rate update might additionally deter attackers.

Nonetheless, they discussed the issue internally and considered fixing this by temporarily disabling deposits in the short period between epoch boundary and exchange rate update. Ultimately, they decided against this. If an attack of this type is ever noticed, the potential effect is minimal, and it could be immediately prevented from re-occurring by deploying the fix suggested above.

References:

- https://github.com/ChorusOne/solido/issues/403

Late Implementation of Unstake Accounts and Denial of Withdrawal (Low)

Severity	Impact	Affected Component
Low	While unstaking, rewards cannot be withdrawn	Unstake Mechanism

At the start of the audit, the mechanism for moving funds from unstake accounts back to the reserve was still unimplemented. Unstake accounts in general seemed incomplete. This was fixed towards the end of the audit.

Perhaps as a consequence of this, a bug was present in the `Withdraw` instruction. In a check at the beginning of `process_withdraw`, total stake, including unstake accounts, is compared with total stake excluding unstake accounts when determining the validator with the highest stake. This meant that if the validator with the most stake had an unstake account with any amount of funds, all withdrawals would fail.

Solution The mechanisms for unstake accounts were implemented toward the end of the audit time-frame and subsequently reviewed by Neodyme. However, the review was kept brief due to limited time.

This specific bug in the `Withdraw` instruction was fixed immediately.

References:

- https://github.com/ChorusOne/solido/pull/391
- https://github.com/ChorusOne/solido/pull/393

Missing Owner Checks in Account Validation (Info)

Severity	Impact	Affected Component
Info	-	Account Data Validation

The account validation routines (in particular, `PartialVoteState::deserialize` and `check_is_st_sol_account`) do not check the owner of the passed accounts. This is relevant, for example, for the validator `stSOL` reward accounts or the validator vote accounts. Normally, these accounts are owned by the Vote Program or the Token Program. However, the validation routines are missing this owner check and only verify that the account's data has the correct format.

A malicious party could create a fake vote account or SPL account and pass it to the contract. As soon as it is stored in the contract, it could use its ownership of these accounts to delete them, creating a new account at the same address. For example, this means that a malicious validator could get the contract to accept a vote account that does not adhere to the vote account parameters that the contract expects.

As this is only relevant for newly added accounts, which should be verified by the DAO anyways, we consider this finding `Info`.

Solution The missing ownership checks were added.

References:

- https://github.com/ChorusOne/solido/issues/401
- https://github.com/ChorusOne/solido/pull/411

Missing Checks on Mint (Info)

Severity	Impact	Affected Component
Info	Malicious solido instance can lock tokens	Initialize/Mint Verification

The contract does not check the `freeze_authority` of the `stSOL` mint. A malicious Lido manager could use this to create a malicious Lido instance and convince people to deposit their `SOL`. The users would receive `stSOL` from the corrupted mint. The malicious manager could then freeze the user-held `stSOL` via the attacker-controlled `freeze_authority`, hence preventing users from withdrawing their `SOL`.

Note that this only applies to new Lido instances created by untrusted parties.

Solution Lido stated that they do not see non-official instances of the contract as a problem and that the manager is assumed to be trusted for the official instance. Additionally, the information on the `freeze_authority` is available publicly on the blockchain and can hence be verified.

References:

- https://github.com/ChorusOne/solido/issues/402

Config Accounts are not Program Derived Accounts (Info)

Severity	Impact	Affected Component
Info	Increased Maintenance burden	Configuration Account

As the solido contract only ever needs to support a single instance, the config account could have been made a singleton. This would prevent attacks, in which a new instance of the contract is re-initialized with overlapping accounts, enabling cross-instance attacks.

A good way to create such a singleton config is to use a *Program Derived Account* (PDA) with hardcoded seeds. However, the current implementation is just as secure, as the Lido team carefully designed the contract to avoid these cross-instance attacks. It does create a small maintenance burden, though, since all instructions added in the future must also be resistant.

Solution The Lido team agreed that this would have been a better choice. However, due to limited time before deployment, they prefer not to make such a sweeping change to the contract. It would also make a re-audit of the new contract necessary, which would significantly delay their launch. Since it currently poses no threat to the security of the contract, they prefer to keep the current design.

Neodyme AG

Dirnismaning 55
Halle 13
85748 Garching
E-Mail: contact@neodyme.io

https://neodyme.io