Security Audit – Lido on Solana v2

Neodyme AG

October 13, 2022

Contents

Introduction

P2P engaged Neodyme to do a detailed security analysis of a new version of the Lido on-chain program. A thorough audit was done between the 22nd of August and the 9th of September 2022.

The audit revealed one medium-severity issue and one low-severity issue; both are resolved.

The following report describes all findings in detail.

Project Overview

Lido is a multi-chain liquid staking protocol. Since September 2021, the Solana blockchain has been supported via its smart contract called `solido`.

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.

Neodyme has audited v1 of Lido before its release. Recently the Lido DAO voted to update the protocol to Version 2, which supports the inclusion of public validators and enables the inclusion of a higher count of them[1].

This updated contract is the target of this audit.

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/lidofinance/solido
- Documentation: https://docs.solana.lido.fi/

Scope

The audit included the full, updated release-candidate for the on-chain program of Lido v2. Specifically, the `migrate_v2` branch of solido at commit `db63de82bee26735b48f10150fac8a1a31177a00` was in scope. All fixes and mitigations of our findings are included in the later commit `c064ee88160a58259c2cb5d03aadd41ed4fb569e`. In addition, some minor nitpicks were later fixed in commit `a1b4d54efcec6319fff883680ced778ae2c70e1e`.

The audit includes both a code security and a financial logic assessment.

[1] https://blog.lido.fi/new-era-for-lido-on-solana/

Methodology

Neodyme's audit team, which consists of security engineers with extensive experience in Solana smart contract security, reviewed 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 that 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,
- Checking for replay protection.

Findings

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

Similar to Lido V1, the design of the updated version is overall quite safe. Even though one of the found bugs resulted in a powerful attack primitive, it was not possible to withdraw user funds. Relevant for the protection of funds is a reduced subset of the program: deposit, withdraw, and the exchange rate.

Contrary to Lido V1, the new version now has multiple accounts that store data on-chain: The main config account is linked to a maintainer and a validator list. This allows Lido to increase the number of supported validators and maintainers in exchange for making the on-chain program a bit more complicated.

Lido continues to have an excellent test suite and contains many helpful comments.

There are four types of authorities to consider when thinking about Lido:

- *Upgrade Authority*: The Lido contract is upgradeable at any time by the upgrade authority. It thus holds full control over all funds. On mainnet, Lido currently uses a multisig smart contract with authority `GQ3QPrB1RHPRr4Reen772WrMZkHcFM4DL5q44x1BBTFm`. Lido is very transparent and outlines all participants of the multisig in their Documentation.
- *Manager*: The manager has the authority to add and remove validators and maintainers, as well as set the reward distribution parameters. He cannot withdraw users' funds but can prevent rewards from being generated. Lido's mainnet deployment uses the same manager as upgrade authority.
- *Maintainer*: Maintainers have the task of ensuring stake is equally distributed among all validators. There are some checks in the smart contract to ensure only the lowest/highest staked validator gets staked/unstaked, but the active balancing is computed and triggered by maintainers. Maintainers can't reduce a user's funds, but they must be trusted, as they could cause rewards to be lower than optimal.
- *User*: Anyone can deposit Sol and get stSol in return. Withdrawal is possible by burning stSol and receiving an active stake account.

Lido V2 also contains a migration instruction to upgrade an existing V1 instance to V2.

All findings are classified in one of four severity levels:

- **Critical**: Bugs that will likely cause a loss of funds. This means that an attacker can trigger them with little or no preparation or even accidentally. Effects are difficult to undo after they are detected.
- **High**: Bugs which can be used to set up a loss of funds in a more limited capacity, or to render the contract unusable.
- **Medium**: Bugs that do not cause a direct loss of funds but lead to other exploitable mechanisms.
- **Low**: Bugs that do not have a significant immediate impact and could be fixed easily after detection.

Name	Severity	Status
Privilege Escalation through Account Type Confusion	Medium	Resolved
Permanent Stake Denial via Stake Account Reuse	Low	Resolved
Checks During Initialization can be Improved	Informational	Resolved
Developer/Treasury Token Account Owners Can Stop Rewards Collection	Informational	Acknowledged

Privilege Escalation through Account Type Confusion (Medium; Resolved)

Severity	Impact	Affected Component	Status
Medium	DoS	Lido Configuration	Resolved

The previous version of Lido only had one account type, the main config account. This also means the old version did not have any explicit account type tags. The new version introduces two additional accounts: a Validator List and a Maintainer List, requiring the introduction of such tags.

Lido v2 does indeed introduce these tags via `AccountType` fields, but the implementation was flawed.

Impact

This bug allows an attacker to add new validators, deactivate existing ones or add and remove maintainers. The attacker cannot execute any reward distribution changes.

As a result, an attacker can temporarily remove all maintainers and validators, thus locking up users' funds. The attacker can also add his own validator and stake it, hoping to claim part of the generated rewards via validator commission. However, the attacker can not keep all of the rewards, as the program only accepts validators that do not exceed the configured commission threshold.

No funds are permanently lost.

As the original manager retains access and can undo everything in practice, this would likely just lead to a loss of rewards in the epochs it takes until the Lido program can be upgraded.

Since some key-grinding is necessary for this attack to work, the estimated cost to execute is a minimal of ~50k USD.

Even though this bug allows an attacker to arbitrarily add, or remove maintainers and validators, we classify it as Medium, since it doesn't directly lead to a loss of funds apart from missed rewards, and the original manager can still act on the instance.

Technical Details

It's usually best to put the type at the very first byte of each account, which Lido did not do. In both of the list-type accounts, the type was at byte offset 6, while in the main Lido struct, it was at offset 2. As Lido deserialization uses `try_from_slice_unchecked`, the length of the account is no implicit tag either.

With some setup, this allows type confusion to occur.

It is possible to create a fake Lido instance with the same validator and maintainer lists as a legit instance.

By doing so, an attacker, who is the manager of the fake Lido instance, can change the original validator and maintainer lists, which should only be possible by the original instance maintainer.

To see that this confusion is possible, we have to compare the List struct to the Lido struct and see if it is possible for them to be similar enough that one can pass for the other:

```
struct List<Maintainer>   -> struct Lido
pub max_entries_0: u8,    -> Lido_version: u8, aka 1
pub max_entries_1: u8,    -> account_type: u8, aka 1 for Lido
pub max_entries_2: u8,    -> 0 (manager pubkey part, grindable)
pub max_entries_3: u8,    -> 0 (manager pubkey part, grindable)
pub Lido_version: u8,     -> 1 (manager pubkey part, grindable)
pub account_type: u8,     -> 3 (manager pubkey part, grindable)
pub current_entries: u32, -> ? (manager pubkey part, grindable)
pub 32-byte-pubkey[0x0101]-> whatever (pubkey fully controlled)
```

As you can see, for this to work, the first few bytes of the manager's pubkey must be fixed to set values. This is possible by generating random keys until you find one which matches the requirements.

Attack Outline:

- Create a new solido instance, with an attacker-controlled key as manager.
- Create a maintainer list with 257 entries, so the first two bytes are 0x01.
- Add a number of maintainers, which can have arbitrary pubkeys, which are just raw bytes.
- Construct that arbitrary data so that the maintainer list looks like a normal Lido account. This needs a lot of grinding (~6.x bytes) for the manager key to be valid, but that's doable by utilizing cloud computing resources.
- Construct the fake Lido account so that it shadows legit validator and maintainer lists. This is possible since the attacker is just writing raw bytes via the maintainer keys, and there are no additional checks.
- Using the fake Lido account, where the attacker is the manager, he can manipulate the legit validator and maintainer lists. This is possible using the normal `add_maintainer` calls etc.
- The attacker can now use the manipulated validator and maintainer lists with the legit Lido account to, for example, change stake distribution.

This is not directly a Loss-of-funds bug since even the manager, or maintainer can't steal staked funds. And the necessary manager-key grinding makes it expensive as well, ~50k$ compute costs in our very

rough estimate. But an attacker can unstake and remove all existing validators and add a new one with a high commission. For the attacker, it could also be interesting just to hijack some stake to run other attacks on the network.

As funds can only be withdrawn from active stake accounts, the attacker could temporarily prevent any withdrawals by deactivating and unstaking all validators.

Resolution

Lido fixed this bug by reordering the `accountType` field to always be the first byte in the account.

Now the only collision possible is with the old Lido instance, which did not have a tag yet and is zero in the first byte, with an empty account.

The only two functions that potentially accept empty accounts are Initialize and Migrate, both of which check enough bytes of the supposedly empty accounts to prevent account confusion there as well.

Fix: Commit

Permanent Stake Denial via Stake Account Reuse (Low; Resolved)

Severity	Impact	Affected Component	Status
Low	DoS	Staking	Resolved

The current design of the Maintainer-only stake deposit instruction has a denial-of-service issue.

Impact

Any maintainer can bring one or more validators into an invalid state, which will block staking for the affected validators.

Note that Lido roughly enforces a uniform stake distribution across all validators. Since the affected validators can never receive additional stake, this will also block all other validators from ever receiving more stake than the affected validators.

In practice, this means that until the affected validators are removed, the total amount staked can not increase, even if more funds are deposited.

Usually, all a maintainer can do is reduce rewards for one epoch, after which the manager can remove the offending maintainer and continue as normal. This bug is slightly different in that the effects are permanent: The affected validators' states will remain broken even after they are completely unstaked and removed. Re-adding the validators even after the offending maintainer is removed will still cause the same issue to occur.

Maintainers are usually trusted but could still misbehave, for example, in cases of a leaked private key or a hack.

As the impact here is limited and only maintainers can trigger it, we rate this bug as low severity. But it can permanently prevent maintainer-chosen validators from participating in Lido.

Technical Details

The root cause of this bug is the reuse of stake accounts, which often makes denial-of-service attacks possible.

This is caused by the fact that accounts are only deleted at the end of a transaction, which is an issue if you reuse stake accounts:

> Some programs may attempt to deinitialize an account by setting its lamports to zero, with the assumption that the runtime will delete the account. This assumption may be valid between transactions, but it is not between instructions or cross-program invocations. (from Solana Documentation - Transactions)

In Lido, a validator can have multiple stake accounts, which are kept track of using a ring-buffer of seeds. Each seed is a `u64`, and at all times, all existing stake accounts are in a range of seeds without gaps, recorded by a `seeds.begin` and `seeds.end` variable.

If a validator is staked for the first time in an epoch, a new stake account with the current `seeds.end` is created. All subsequent staking operations in the same epoch will create a temporary stake account and then immediately merge it into the previously existing stake account for the same epoch. This minimizes the number of stake accounts.

However, this temporary stake account also uses `seeds.end`, which means that the same stake account will be reused later. Now consider the following attack:

1. A malicious maintainer stakes 1 Sol to a minimum-stake validator.
    - A stake account using `seeds.end` is created and delegated.
    - Afterwards, `seeds.end` is incremented by one.
2. The maintainer stakes another 1 Sol to the same still-minimum-stake validator.
    - A second stake account using the incremented `seeds.end` is created
    - The second stake account is merged into the first stake account. This reduces the second account's lamports to zero.
    - `seeds.end` is NOT incremented, as the second account will be deleted at the end of the transaction.
3. The maintainer places a third instruction in the same transaction, transferring the required rent into the second account.
    - Since this instruction runs in the same transaction, the second account has not yet been deleted, is still owned by the stake program, and still has data.
    - The `seeds.end` account now has the required rent to avoid deletion.

The stake program takes care not to be vulnerable against these revive-attacks by resetting the account back to `Stake::Uninitialized`. But the account is still there, owned by the stake program and has data allocated. Even worse, anyone can now initialize this account with their own stake/withdraw authority, as the stake program doesn't require signatures for initialization. This means Lido, even though it "owns" the PDA, will never be able to delete the `seeds.end` account.

This, in turn, means all future attempts to stake this validator will fail, as Lido will be unable to allocate/assign the required `seeds.end` account. As the validator is a minimum-stake-account, deposits to other validators aren't allowed either.

If there are multiple minimum-stake-validators when beginning the attack, the malicious maintainer can corrupt the state for all of them.

Only depositing new stake is affected; everything else still works as intended. In particular, deactivating a validator, unstaking, and withdrawing are unaffected.

Quick Fix

As a quick fix, the manager can remove the maintainer and then deactivate all broken validators. As the affected validators' state will stay broken, they should never be re-added, but everything else will work as before.

Related issue: Re-adding a removed validator causes seeds to reset

If the manager ever removes a validator and later adds it back in, the seeds will start back at 0. This will cause the same stake accounts to be reused and makes the effects of the attack explained above permanent.

An ahead-looking maintainer could also revive an unstake account in a similar fashion. This would make it impossible to unstake the affected validator if it was ever re-added and staked. However, the maintainer would have to do so while the validator was still added and staked.

During discussion of this finding with P2P, the following suggestion emerged: This could be fixed by adding an additional global counter, say a "validator generation". Each time a validator is added, the current value of the global generation counter is copied to the validator. Afterwards, the global counter is incremented by one. If this generation value is included in all relevant PDA seeds, then re-adding a validator will not cause stake accounts to be reused. Neodyme agrees with this suggestion.

Alternatively, the manager could choose the starting range for the seeds when adding a validator.

Both methods allow re-adding any validator, no matter what happened to its old stake PDAs.

Resolution

Lido now uses a different temporary account tied to the current epoch while merging stake. This means if a maintainer ever behaves that way, only one epoch of rewards will be affected and Lido will self-repair after that.

Fix: Commit

Checks During Initialization can be Improved (Informational; Resolved)

Severity	Impact	Affected Component	Status
Informational	/	Initialization	Resolved

There are a few checks in initialize that don't quite do what they are intended to do.

First, `check_mint()` only checks the `mint_authority`, but mints have another authority: the `freeze_authority`. Lido has to check it as well; otherwise, you can initialize an instance with a mint where an arbitrary freeze authority is set. This would allow the initializer to arbitrarily freeze any stSol token account.

Second, initialization requires passing the `reserve_account` as an account. However, Lido doesn't check that the passed account actually matches the expected PDA. At the moment, the passed account is only checked to be rent-exempt. Due to the missing key check, you can construct a Lido instance where the reserve is *not* rent-exempt. However, this has negligible impact. Lido already calculates the reserve account PDA via seeds in order to store the bump seed. The initialization can thus easily be extended to also check the key of the passed reserve account.

Third, `check_mint()` is missing an owner check for the mint account. That could allow an attacker to craft a mint that has `supply != 0`, get it in a Lido instance with some trickery, and then mint himself a lot of tokens before giving the authority to Lido. In practice, this attack is not possible, as the mint is very implicitly owner-checked by the existence of the `treasury_account`, which has to be a valid spl-token account for this mint, which is only possible if the mint is owned by the spl-token program as-well. But an explicit owner-check would make this a lot clearer.

We only classify this finding as informational, as the current Lido instance is correctly initialized and thus not vulnerable to these small initialization issues.

Resolution

Lido now checks the freeze authority and owner of the mint in `check_mint()`. The `reserve_account` PDA is also checked at initialization.

Fix: Commit

Developer/Treasury Token Account Owners Can Stop Rewards Collection (Informational; Acknowledged)

Severity	Impact	Affected Component	Status
Informational	/	Reward Distribution	Acknowledged

The current implementation uses a push design for handing out rewards to the treasury and developer. Whenever rewards are to be distributed, the program actively transfers funds to the treasury and developer. A pull-based design is usually the better choice because it sidesteps Denial of Service attacks.

In solido, the initialization checks that treasury and developer accounts are valid stSol token accounts. This is a nice sanity check but doesn't give you any certainty since the respective owners can, at any time, just close the token account. This will make all future transfers to it fail.

This gives both developer and treasury account owners some unintended authority over the protocol. By closing their accounts, they can make reward distribution impossible, and thus, in turn, all reward-collection by Lido.

We only classify this finding as informational since, in practice, the manager can unblock the protocol by changing the affected account with `ChangeRewardDistribution`. No rewards are permanently lost; they are just delayed until they can be collected after the unblock.

Neodyme AG

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

https://neodyme.io