Security Audit – Anker on Solana

Neodyme

2022-04-06

Contents

Project Summary

Anker engaged Neodyme to do a detailed security analysis of their Solana on-chain program. A thorough audit was done from January 10th to February 18st.

Target of the audit was the source-code of the Anker on-chain program version `1.2.0`, specifically the commit hash 0603c66bb366d447a946e4121a30ee0ad5444455.

The code has been written to a high standard, and the team expertly responded to all questions. They also provide a detailed Security Documentation explaining the security relevant functionality and threat model.

Nevertheless, the audit revealed two major vulnerabilities, which were reported and subsequently fixed by the Anker team. This report describes these findings in detail.

Introduction to Anker

Anker is the Solana intergration for Anchor, a cross chain yield aggregation platform on Terra. The Solana component is built atop solido and is the sole target of this audit.

Anker has a three main functions, as described in the Security Documentation:

- **Deposit**, where users deposit stSOL and receive bSOL in return.
- **Withdraw**, where users redeem their stSOL by returning the bSOL.
- **Claim Rewards**, where, if the value of the stSOL in the reserve is greater than what is needed to back the bSOL supply at a 1 bSOL = 1 SOL exchange rate, the program can swap the excess stSOL for UST against an AMM, and it sends the proceeds through Wormhole to a preconfigured address on Terra.

In addition, it also has a few privileged instruction, only accessable to the manager. They allow them to change the token swap pool used to sell of stSol and change the address UST get transferred to on the terra side.

The Anker 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/tree/main/anker
- Anchor Documentation: https://docs.anchorprotocol.com/
- Anker Security Documentation: https://github.com/ChorusOne/solido/blob/main/anker/SECURITY.md

Methodology

Neodyme's audit team performed a comprehensive examination of the Anker 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 Anker'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 Anker chose one of the safest ones: there is a single Anker instance per solido instance. The corresponding solido instance is used in the derivation of the Anker program-derived address, thus making all attacks that work by confusing between multiple program-owned accounts impossible. This approach has the minor drawback that all Anker 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. Additionally, this contract implements privileged instructions, accessable to the manager, allowing them to change the UST rewards destination on Terra. As Anker has not been deployed to the Solana-Mainnet yet, we cannot make any assertions as to the contract-upgradability or manager key integrety. However, unlike many other projects, Anker already has a distributed set of trusted owners, which they use in a multisig contract. The multisig and its members are documented publicly and can be inspected using the solido command line utility:

```
    ~$ ./target/debug/solido \
        --cluster https://api.mainnet-beta.solana.com \
        multisig show-multisig \
        --multisig-address 3cXyJbjoAUNLpQsFrFJTTTp8GD3uPeabYbsCVobkQpD1 \
        --multisig-program-id AAHT26ecV3FEeFmL2gDZW6FfEqjPkghHbAkNZGqwT8Ww

    Program derived address: GQ3QPrB1RHPRr4Reen772WrMZkHcFM4DL5q44x1BBTFm
    Threshold: 4 out of 7
    Owners:
      Cv6GM219kzMrdUUdgDGVJUPW6fGosvrhsFrvmEhz3Mc6
      ENH1xvwjinUWkwEgw1hKduyAg7CrJMiKvr9nAS7wLHrp
      6CawqfAJDviZGfUpHFJgeauq6H9vhKuivMMZULZeGnPw
      F4VFp4tFTyrQWo9YVjCbPE5eQP27ice2zyGDp2tN2Rkm
      AnoVUukL1fMAwEp4y2rrZV45BNHLes8ZwWsCRgEwhGH4
      6S21QCmpAadEhHj3pY2RMbPMGwgYNvS4Pd7zUXoRDMdK
      DHLXnJdACTY83yKwnUkeoDjqi4QBbsYGa1v8tJL76ViX
```

They assured us that this *decentralized autonomous organization* (DAO) will be controlling both the upgrade authority and the manager key. This provides a higher level of protection, as the DAO has to approve any modifications, ruling out a single-party rug-pull by the developer.

All findings are classified in one of four severity levels:

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

In total, there was one `critical` and one `high` finding.

Name	Severity	Status
Missing Token Program validation in SellRewards	Critical	Resolved
Sandwiching issue in SellRewards	High	Resolved

Missing Token Program validation in SellRewards (Critical; Resolved)

Users can deposit their stSol in order to get bSol and Anker maintains an stSol reserve to back the bSOL supply at a 1 bSOL = 1 SOL exchange rate. However since stSol appreciate in value, Anker has been fitted with a SellRewards instruction to sell off any exess stSol value build-up. This instruction is supposed to be called by crankers and therefore is permissionless. Anker uses Orca's deployment of SPL token-swap program to swap stSol for UST.

The issue is that SellRewards never verifies that it is actually dealing with Orca's token-swap program, allowing an attacker to supply an arbitrary program. This program will subsequently be called using Solana's invoke_signed() API and signed by the token_swap_authority. An attacker can craft their own malicious fork of the SPL token-swap program, that withdraws all the funds from the reserve to an attacker controlled account.

```rust
pub fn swap_rewards(
    program_id: &Pubkey,
    amount: StLamports,
    anker: &Anker,
    accounts: &SellRewardsAccountsInfo,
) -> ProgramResult {
    // [...]

    /* !!! Neodyme !!!
     *
     * check_token_swap doesn't verify the token_swap_program_id pubkey
     *
     */
    anker.check_token_swap(program_id, accounts)?;

    let swap_instruction = spl_token_swap::instruction::swap(
        accounts.token_swap_program_id.key,
        /* !!! Neodyme !!!
         *
         * Unchecked token_swap_program_id account gets
         * used as program_id for a swap instruction
         *
         */
        // [...]
    )?;

    let authority_signature_seeds = [
        &accounts.anker.key.to_bytes(),
        ANKER_RESERVE_AUTHORITY,
        &[anker.reserve_authority_bump_seed],
    ];
    let signers = [&authority_signature_seeds[..]];
    /* !!! Neodyme !!!
     *
```

```
         * ANKER_RESERVE_AUTHORITY signs the CPI to an
         * attacker controlled program
         *
         */
        invoke_signed(
            &swap_instruction,
            &[
                // [...]
            ],
            &signers,
        )
    }
```

Solution The Anker team responded immediately by adding a regression test and adding the missing check. Neodyme verified the fix.

References: https://github.com/ChorusOne/solido/pull/512

Sandwiching issue in SellRewards (High; Resolved)

Anker sells off exess stSol by using an Orca stSol<->UST token swap and sending the resulting UST through Wormhole to a rewards account on the Terra side. Lido's exchange rate is used to compute rewards for an epoch and the swap gets performed at the start of the next epoch.

The full amount is swapped in a single operation without slippage protection, which could lead to a lot of slippage in a low liquidity pool. An even bigger issue however is that since this instruction is intended to be called by crankers, a malicious cranker can exploit this by sandwiching the sale between two of their own swaps to steal the rewards.

```rust
fn process_sell_rewards(program_id: &Pubkey, accounts_raw: &[
    AccountInfo]) -> ProgramResult {

    // [...]

    // Get StLamports corresponding to the amount of b_sol minted.
    let st_sol_amount = solido.exchange_rate.exchange_sol(Lamports(
        b_sol_supply))?;

    /* !!! Neodyme !!!
     *
     * Lido's exchange rate is used to determine the
     * amout of rewards per epoch
     *
     */

    // If `reserve_st_sol` < `st_sol_amount` something went wrong, and
        we abort the transaction.
    let rewards = (reserve_st_sol - st_sol_amount)?;


    /* !!! Neodyme !!!
     *
     * swap_rewards will swap the full rewards on an orca stSol<->UST
     * pool in a singe trade with no slippage protection
     *
     */
    swap_rewards(program_id, rewards, &anker, &accounts)?;

    // [...]

    msg!("Swapped {} for {}.", st_sol_amount, swapped_ust);

    anker
        .metrics
        .observe_token_swap(st_sol_amount, swapped_ust)?;
    anker.save(accounts.anker)
}
```

Solution The Anker team responded immediately by adding slippage protection based on recent prices. The way it works is nicely described by a comment in the fix:

> Next are three constants related to stored stSOL/UST prices. Because Anker is permissionless, everybody can call `SellRewards` if there are rewards to sell. This means that the caller could sandwich the `SellRewards` between two instructions that swap against the same stSOL/UST pool that Anker uses, to give us a bad price, and take the difference. To mitigate this risk, we set a `min_out` on the swap instruction, but in order to do so, we need a "fair" price. For that, we sample 5 past prices, at least some number of slots apart (enough that they are produced by different leaders), but also not too old, to make sure the price is still fresh. Then we take the median of that as a "fair" price and set `min_out` based on that. Now if anybody is trying to sandwich us, they would also have to sandwich 3 of those 5 times where we sample the price (and they pay swap fees), and they are competing with our honest maintenance bot for that (and possibly with others). Also, having a recent price ensures that we don't sell rewards at times of extreme volatility.

Neodyme verified the fix.

References: https://github.com/ChorusOne/solido/pull/511

Neodyme AG

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

https://neodyme.io