// Public Solana Program Security Assessment

08.05.2024 - 08.09.2024

Restaking

Solayer

HALBORN

Restaking - Solayer

Prepared by: HALBORN

Last Updated 08/13/2024

Date of Engagement by: August 5th, 2024 - August 9th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	0	0	0	0	3

TABLE OF CONTENTS

1. Introduction

Solayer team engaged Halborn to conduct a security assessment on their Restaking **Solana program** beginning on **August 5th, 2024**, and ending on **August, 09th, 2024**. The security assessment was scoped to the Solana Program provided in solayer-labs/restaking-program GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

The Restaking program has both administrative and user-facing instructions, and the main purpose is to allow the deposit of different LST assets (collaterals) in exchange of RST assets, which currently is **exclusively** sSOL.

Administrative Instructions:

- **Initialize:** Allows the initialization of a `pool` PDA, derived from the `lst_mint` (collateral) account address, configures a protocol `vault` to receive the collateral LST, and defines the RST, which is the asset the `Solayer` protocol gives back in exchange for collaterals.
- **Batch Unfreeze:** Utility instruction, used to batch `thaw` accounts.

User-facing instructions:

- **Restake:** Allows users to deposit their LST collateral in the respective `pool` and `vault`, in order to receive (mint) RST tokens from the protocol in exchange. Performs CPI to `mint_to` method in the interfaced token program.
- **Unrestake:** Allows users to withdraw their LST collateral from the respective `pool` and `vault`, and give back (burn) the RST. Performs CPI to `burn` method in the interfaced token program.

2. Assessment Summary

Halborn was provided **4 days** for the engagement and assigned one full-time security engineer to review the security of the Solana Program in scope. The engineer is a blockchain and smart contract security expert with advanced smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the `Restaking` Solana Program.
- Ensure that the program's functionality operates as intended.

In summary, Halborn identified some non-critical issues, that were addressed and acknowledged by the `Solayer team`:

- Avoid using `require!` inside loops
- Lack of Zero Amount validation
- Decimals should be enforced
- Outdated dependencies

Overall, the program in-scope is adherent to Solana's best-practices and carries consistent code quality.

3. Test Approach And Methodology

Halborn performed a combination of a manual review of the source code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the program assessment. While manual testing is recommended to uncover flaws in business logic, processes, and implementation; automated testing techniques help enhance coverage of programs and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual program source code review to identify business logic issues.
- Mapping out possible attack vectors.
- Thorough assessment of safety and usage of critical Rust variables and functions in scope that could lead to arithmetic vulnerabilities.
- Scanning dependencies for known vulnerabilities (`cargo audit`).
- Local runtime testing (`anchor test`).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: restaking-program

(b) Assessed Commit ID: 2cf4745

(c) Items in scope:

- ./lib.rs
- ./errors.rs
- ./contexts/initialize.rs
- ./contexts/restaking.rs
- ./contexts/mod.rs
- ./contexts/batchunfreeze.rs
- ./constants.rs
- ./state/mod.rs
- ./state/restaking_pool.rs

Out-of-Scope:

REMEDIATION COMMIT ID:

- 46c0907

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	0	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
AVOID USING REQUIRE! INSIDE LOOPS	INFORMATIONAL	ACKNOWLEDGED
LACK OF ZERO AMOUNT VALIDATION	INFORMATIONAL	ACKNOWLEDGED
OUTDATED DEPENDENCIES	INFORMATIONAL	SOLVED - 08/12/2024

7. FINDINGS & TECH DETAILS

7.1 AVOID USING REQUIRE! INSIDE LOOPS

// INFORMATIONAL

Description

The current implementation of the `BatchUnfreeze` instruction uses `require!` statements inside loops, which can lead to inefficiencies and potential failures. If a `require!` statement fails within a loop, the entire loop will terminate, causing the entire operation to fail.

- `programs/restaking-program/src/contexts/batchunfreeze.rs`

```
    pub fn batch_thaw_lst_accounts(&mut self, lst_token_accounts: &[Accou
        if self.signer.key() != Pubkey::from_str(SOLAYER_ADMIN).unwrap()
            return Err(ProgramError::MissingRequiredSignature.into());
        }
        for unchecked_lst_token_account in lst_token_accounts.iter() {
            // check if the account is owned by the token program before
            require!(unchecked_lst_token_account.owner.key() == self.toke
            let token_account = TokenAccount::try_deserialize(&mut &unche
            require!(token_account.mint == self.rst_mint.key(), Restaking
            if !token_account.is_frozen() {
                continue;
            }
            let bump = [self.pool.bump];

            let signer_seeds: [&[&[u8]]; 1] = [
                &[
                    b"pool",
                    self.lst_mint.to_account_info().key.as_ref(),
                    &bump
                ][..]
            ];

            let ctx = CpiContext::new_with_signer(
                self.token_program.to_account_info(),
                ThawAccount {
                    account: unchecked_lst_token_account.clone(),
                    authority : self.pool.to_account_info(),
                    mint: self.rst_mint.to_account_info()
                },
                &signer_seeds
```

```
                );

                thaw_account(ctx)?;
            }
            Ok(())
        }
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

It is recommended to avoid using `require!` statements inside loops, in order to prevent undesired failures.

Remediation Plan

ACKNOWLEDGED: The **Solayer team** acknowledged this issue.

7.2 LACK OF ZERO AMOUNT VALIDATION

// INFORMATIONAL

Description

The program in-scope does not prevent the `restake` and `unrestake` methods from being called with `amount == 0`.

- `programs/restaking-program/src/contexts/restaking.rs`

```
    pub fn restake(ctx: Context<Restaking>, amount: u64) -> Result<()> {
        // Check if solayer_signer has signed the restake transaction
        // since we will impose TVLs caps at different epochs
        /*let solayer_signer: &UncheckedAccount<'_> = &ctx.accounts.solay
        if !solayer_signer.is_signer {
            return Err(ProgramError::MissingRequiredSignature.into());
        }*/

        ctx.accounts.thaw_rst_account()?;
        ctx.accounts.stake(amount)?;
        ctx.accounts.mint_rst(amount)?;
        // Check if RST mints should be frozen
        if !is_liquid_rst_mints(&ctx.accounts.rst_mint.key()) {
            ctx.accounts.freeze_rst_account()?;
        }
        Ok(())
    }

    pub fn unrestake(ctx: Context<Restaking>, amount: u64) -> Result<()>
        ctx.accounts.thaw_rst_account()?;
        ctx.accounts.unstake(amount)?;
        ctx.accounts.burn_rst(amount)?;
        // Check if RST mints should be frozen
        if !is_liquid_rst_mints(&ctx.accounts.rst_mint.key()) {
            ctx.accounts.freeze_rst_account()?;
        }
        Ok(())
    }
```

The entry-point functions in `lib.rs` does not handle this verification either. While this condition does not lead to immediate financial loss, it should be checked to keep overall consistency.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

Consider adding a verification before the execution of the `restake` and `unrestake` methods, blocking operations with `amount == 0`.

Remediation Plan

ACKNOWLEDGED: The **Solayer team** acknowledged this issue.

7.3 OUTDATED DEPENDENCIES

// INFORMATIONAL

Description

It was identifying during the assessment of the program `restaking` in-scope that its dependencies for the Anchor framework and also for Solana are not current.

```
[[package]]
name = "solana-program"
version = "1.18.7"
```

```
[[package]]
name = "anchor-lang"
version = "0.29.0"
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

It is recommended to update dependencies to their current versions, as specified:

- Solana: `v1.18.20`
- Anchor: `v0.31.0`

Remediation Plan

SOLVED: The **Solayer team** has solved this issue as recommended. The commit hash containing the modification is `46c09073a6dad390f435dc76f17e35849f2c6d1b`.

Remediation Hash

https://github.com/solayer-labs/restaking-program/commit/46c09073a6dad390f435dc76f17e35849f2c6d1b

8. AUTOMATED TESTING

STATIC ANALYSIS REPORT

Description

Halborn used automated security scanners to assist with detection of well-known security issues and vulnerabilities. Among the tools used was `cargo audit`, a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in `https://crates.io` are stored in a repository named The RustSec Advisory Database. `cargo audit` is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. All vulnerabilities shown here were already disclosed in the above report. However, to better assist the developers maintaining this code, the auditors are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.

Cargo Audit Results

ID	CRATE	DESCCRIPTION
RUSTSEC-2022-0093	ed25519-dalek	Double Public Key Signing Function Oracle Attack on `ed255109-dalek`
RUSTSEC-2024-0344	curve25519-dalek	Timing variability in `curve25519-dalek`'s Scalar29::sub/Scalar52::sub
RUSTSEC-2021-0145	atty	Potential unaligned read

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.