// Security Assessment

02.13.2025 - 02.21.2025

Solana Staker

TruFin

HALBORN

Solana Staker - TruFin

Prepared by: HALBORN

Last Updated 03/06/2025

Date of Engagement by: February 13th, 2025 - February 21st, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	0	0	0	2	1

TABLE OF CONTENTS

1. Introduction

TruFin engaged Halborn to conduct a security assessment on their Staker **Solana program** beginning on **February 14th, 2025**, and ending on **February, 26th, 2025**. The security assessment was scoped to the Solana program provided in TruFin-io/solana-staker-hb GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

The staker program is a staking vault solution built to streamline SOL staking - users deposit SOL into a stake pool and receive a reward-bearing token (TruSOL) which entitles them to redeem their staked SOL. The vault either lets users pick which validator to delegate their stake to or auto-manages validator allocations for enhanced performance.

To maintain compliance and security, every depositor must be allowlisted. The program checks a user's whitelist status at deposit time. In addition, the vault owner can pause deposits in emergencies, replace the stake manager authority, and add or remove validators from the pool.

2. Assessment Summary

Halborn was provided **1 week, 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 staker Solana Program.
- Ensure that the program's functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were accepted and acknowledged by the TruFin team:

- Explicitly verify the signer's key is the owner's key (rather than just any agent). [Risk Accepted]
- Ensure that the InitializeStaker instruction is called by a trusted and known address, such as the program's upgrade authority. [Risk Accepted]
- Require multiple signers in stake reallocation operations. [Acknowledged]

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 (`solana-test-framework`).

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:

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

EXPLOITABILITY 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

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
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
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: solana-staker-hb

(b) Assessed Commit ID: a91e9b5

(c) Items in scope:

- src/instructions/whitelist.rs
- src/instructions/validators.rs
- src/instructions/setters.rs
- src/instructions/initialize.rs
- src/instructions/staking.rs
- src/instructions/mod.rs
- src/constants.rs
- src/error.rs
- src/lib.rs
- src/state/types.rs
- src/state/events.rs
- src/state/mod.rs

Out-of-Scope: Third-party dependencies and economic attacks.

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	2

INFORMATIONAL
1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
EXCESSIVE TRUST ASSUMPTION ON AGENTS	LOW	RISK ACCEPTED - 03/03/2025
STAKER INITIALIZATION CAN BE FRONT-RUN	LOW	RISK ACCEPTED - 03/03/2025
SINGLE-SIGNATURE AUTHORITY FOR STAKE REALLOCATIONS	INFORMATIONAL	ACKNOWLEDGED - 03/03/2025

7. FINDINGS & TECH DETAILS

7.1 EXCESSIVE TRUST ASSUMPTION ON AGENTS

// LOW

Description

The `AddAgent` and `RemoveAgent` instructions only require that the caller is already recognized as an existing `Agent` account. No additional check (e.g., "**is owner?**") is enforced before creating new agents or removing them. Practically, if one agent is ever compromised or acts maliciously, the attacker can spawn more malicious agents or remove legitimate ones—bypassing any need for the actual owner's key.

Code Location:

- solana-staker-hb/programs/staker/src/instructions/whitelist.rs

```
#[derive(Accounts)]
#[event_cpi]
#[instruction(agent: Pubkey)]
pub struct AddAgent<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,

    #[account(
        init,
        payer = signer,
        space = ANCHOR_DISCRIMINATOR + Agent::INIT_SPACE,
        seeds = [b"agent", agent.as_ref()],
        bump
    )]
    pub new_agent_account: Account<'info, Agent>,

    #[account(
        seeds = [b"agent", signer.key().as_ref()],
        bump
    )]
    pub agent_account: Account<'info, Agent>,

    pub system_program: Program<'info, System>,
}

/// Processes the `AddAgent` instruction
pub fn process_add_agent(ctx: Context<AddAgent>, agent: Pubkey) -> Resu
    emit_cpi! {
```

```
        AgentAdded {
            new_agent: agent
        }
    };

    Ok(())
}

#[derive(Accounts)]
#[event_cpi]
#[instruction(agent: Pubkey)]
pub struct RemoveAgent<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,

    #[account(
        mut,
        constraint = agent != access.owner.key() @ ErrorCode::CannotRe
        seeds = [b"agent", agent.as_ref()],
        bump,
        close = signer
    )]
    pub agent_account_to_remove: Account<'info, Agent>,

    #[account(
        seeds = [b"agent", signer.key().as_ref()],
        bump
    )]
    pub agent_account: Account<'info, Agent>,

    pub system_program: Program<'info, System>,

    #[account(
        seeds = [b"access"],
        bump
    )]
    pub access: Account<'info, Access>,
}

/// Processes the `RemoveAgent` instruction
pub fn process_remove_agent(ctx: Context<RemoveAgent>, agent: Pubkey)
    emit_cpi! {
        AgentRemoved {
            removed_agent: agent
        }
    };
```

```
    Ok(())
}
```

Moreover, the instructions for modifying user whitelist status (`AddUserToWhitelist`, `AddUserToBlacklist`, `ClearUserStatus`) require only that the signer be an `Agent`. Thus, any agent—if compromised—can whitelist arbitrary addresses or block legitimate users. This could circumvent AML/KYC intentions or sabotage deposits from real customers.

Code Location:
– solana-staker-hb/programs/staker/src/instructions/whitelist.rs

```
#[derive(Accounts)]
#[event_cpi]
#[instruction(user: Pubkey)]
pub struct AddUserToWhitelist<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,

    #[account(
        init_if_needed,
        constraint = user_whitelist_account.status != WhitelistUserSta
        payer = signer,
        space = ANCHOR_DISCRIMINATOR + UserStatus::INIT_SPACE,
        seeds = [b"user", user.as_ref()],
        bump
    )]
    pub user_whitelist_account: Account<'info, UserStatus>,

    #[account(
        seeds = [b"agent", signer.key().as_ref()],
        bump
    )]
    pub agent_account: Account<'info, Agent>,

    pub system_program: Program<'info, System>,
}

/// Processes the `AddUserToWhitelist` instruction
pub fn process_add_user_to_whitelist(ctx: Context<AddUserToWhitelist>,
    let user_status = &mut ctx.accounts.user_whitelist_account;
    let old_status = user_status.status.clone();
    user_status.status = WhitelistUserStatus::Whitelisted;
    emit_cpi! {
        WhitelistingStatusChanged {
```

```
            user,
            old_status,
            new_status: WhitelistUserStatus::Whitelisted
        }
    };
    Ok(())
}

#[derive(Accounts)]
#[event_cpi]
#[instruction(user: Pubkey)]
pub struct AddUserToBlacklist<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,

    #[account(
        init_if_needed,
        constraint = user_whitelist_account.status != WhitelistUserSta
        payer = signer,
        space = ANCHOR_DISCRIMINATOR + UserStatus::INIT_SPACE,
        seeds = [b"user", user.as_ref()],
        bump
    )]
    pub user_whitelist_account: Account<'info, UserStatus>,

    #[account(
        seeds = [b"agent", signer.key().as_ref()],
        bump
    )]
    pub agent_account: Account<'info, Agent>,

    pub system_program: Program<'info, System>,
}

/// Processes the `AddUserToBlacklist` instruction
pub fn process_add_user_to_blacklist(ctx: Context<AddUserToBlacklist>,
    let user_status = &mut ctx.accounts.user_whitelist_account;
    let old_status = user_status.status.clone();
    user_status.status = WhitelistUserStatus::Blacklisted;
    emit_cpi! {
        WhitelistingStatusChanged {
            user,
            old_status,
            new_status: WhitelistUserStatus::Blacklisted
        }
    };
```

```rust
    Ok(())
}

#[derive(Accounts)]
#[event_cpi]
#[instruction(user: Pubkey)]
pub struct ClearUserStatus<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,

    #[account(
        init_if_needed,
        constraint = user_whitelist_account.status != WhitelistUserSta
        payer = signer,
        space = ANCHOR_DISCRIMINATOR + UserStatus::INIT_SPACE,
        seeds = [b"user", user.as_ref()],
        bump
    )]
    pub user_whitelist_account: Account<'info, UserStatus>,

    #[account(
        seeds = [b"agent", signer.key().as_ref()],
        bump
    )]
    pub agent_account: Account<'info, Agent>,

    pub system_program: Program<'info, System>,
}

/// Processes the `ClearUserStatus` instruction
pub fn process_clear_user_status(ctx: Context<ClearUserStatus>, user:
    let user_status = &mut ctx.accounts.user_whitelist_account;
    let old_status = user_status.status.clone();
    user_status.status = WhitelistUserStatus::None;
    emit_cpi! {
        WhitelistingStatusChanged {
            user,
            old_status,
            new_status: WhitelistUserStatus::None
        }
    };
    Ok(())
}
```

BVSS

AO:S/AC:L/AX:L/R:N/S:C/C:M/A:H/I:H/D:H/Y:H (3.6)

Recommendation

If the intended design is that only the owner can manage agent creation/removal, explicitly verify the signer's key is the owner's key (rather than just any agent). Otherwise, clearly document that any agent has the power to onboard/remove other agents, and ensure operational controls around key management.

Regarding the whitelist mechanism, if more granular or multi-signature control over the whitelist is desired (for compliance, risk, or administrative reasons), enforce either an "owner only" check or a multi-agent threshold. Otherwise, explicitly document that any valid agent may set user statuses.

Remediation Comment

RISK ACCEPTED: The **TruFin team** accepted the risk of this finding.

7.2 STAKER INITIALIZATION CAN BE FRONT-RUN

// LOW

Description

Until `InitializeStaker` runs, there is no `owner` and no `Access` account. The first party to call `InitializeStaker` becomes the de facto owner and creates the “owner’s agent.” This is standard in many on-chain deployments but remains a risk if the deployer does not call it immediately (e.g., a race condition at launch).

Code Location:

– `solana-staker-hb/programs/staker/src/instructions/initialize.rs`

```
pub fn process_initialize_staker(ctx: Context<InitializeStaker>) -> Re
    let access_control = &mut ctx.accounts.access;
    access_control.owner = ctx.accounts.owner_info.key();
    access_control.is_paused = false;
    access_control.stake_manager = ctx.accounts.stake_manager_info.key

    emit_cpi!(StakerInitialized {
        owner: ctx.accounts.owner_info.key(),
        stake_manager: ctx.accounts.stake_manager_info.key(),
    });
    Ok(())
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:C/C:N/A:N/I:L/D:N/Y:N (3.1)

Recommendation

To mitigate these risks, it is recommended to ensure that the `InitializeStaker` instruction is called by a trusted and known address, **such as the program's upgrade authority**, and that proper access controls and validations are in place.

Remediation Comment

RISK ACCEPTED: The **TruFin team** accepted the risk of this finding.

7.3 SINGLE-SIGNATURE AUTHORITY FOR STAKE REALLOCATIONS

// INFORMATIONAL

Description

The stake manager instructions (`AddValidator`, `RemoveValidator`, `IncreaseValidatorStake`, `DecreaseValidatorStake`) allow the designated **“stake manager”** or **“owner”** to reallocate large sums of staked SOL at will. While there is no direct path to withdraw SOL to an arbitrary address, a compromised stake manager key can stake to poor validators or degrade yields for the pool.

Code Location (instructions):

- AddValidator
- RemoveValidator
- IncreaseValidatorStake
- DecreaseValidatorStake

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:M/Y:M (1.3)

Recommendation

Require multiple signers in stake reallocation operations. Additionally, create rules (caps, rate-limits, or additional sign-offs) to mitigate the impact of compromised transactions.
Optionally, utilize a multi-signature wallet for sensitive accounts.

Remediation Comment

ACKNOWLEDGED: The **TruFin team** acknowledged this finding.

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-2024-0093	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

---

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.