Code Audit

for

Nosana

https://opcodes.fr / contact@opcodes.fr

15 Rue des Halles, 75001 Paris, France

Project Information

Project	
Mission	Code audit
Client	Nosana
Start Date	08/01/2022
End Date	08/10/2022

Document Revision			
Version	Date	Details	Authors
1.0	08/09/2022	Document creation	Thibault MARBOUD Baptiste OUERIAGLI
1.1	08/10/2022	Peer review	Guillaume FRANCQUEVILLE

Table of contents

Overview

Mission Context

The purpose of the mission was to perform a code audit to discover issues and vulnerabilities in the mission scope. Comprehensive testing has been performed using automated and manual testing techniques.

Mission Scope

As defined with Nosana prior to the mission, the scope of this assessment was a staking Solana program. The code source was supplied through the following Gitlab repository:

- https://gitlab.com/nosana-ci/apps/platform/spl / a04fd511 (main)

OPCODES engineers were due to strictly respect the perimeter agreed with Nosana as well as respect the ethical hacking behavior.

Note: OPCODES engineers only audited the staking program. Other programs in the repository were not part of the scope.

Project Summary

Nosana is building a democratized CPU cloud computing network. Anyone will be able to rent out their spare CPU to power projects. Nosana especially focuses on running project CI/CD infrastructure.

The scope of this assessment was Nosana's staking program, where users can lock tokens in a vault and claim rewards using the reward program. User's weight is derived from the number of tokens they are locking and the withdraw time. Nosana staking program only supports one pool using their Nosana SPL token. From a technical point of view, OPCODES engineers found Nosana codebase well-structured and designed with responsibilities being spitted between their different programs. For example, the staking system is composed of two programs:

- The staking program: It manages the deposit/withdrawal of the user's funds.
- The reward program (out of scope): It handles the claim of rewards and read from the staking program state.

Segregating the logic between different program is a good practice as a vulnerability in the reward program would not affect the users' funds locked in the staking program. It also makes programs smaller and easier to understand.

Synthesis

Security Level: GOOD

The overall security level is considered as good. OPCODES assessment only produced one medium and two minor results.

The medium vulnerability concerns the slashing privilege. Indeed, the staking program implements a slashing mechanism that is centralized to one authority/address.

The first minor vulnerability regards the fact that only one token account is used to store all the staked tokens. When it comes to users' fund, OPCODES engineers would have preferred to see some segregation. The second minor vulnerability is a harmless overflow in the *extend* instruction.

OPCODES also reported four informational issues that do not lead to any exploitable scenario but may enforce bad practice. They represent possible improvements that could be done to improve the security of the program.

OPCODES noticed a good testing coverage. Indeed, each instruction is tested multiple times with both successful and unsuccessful scenarios. OPCODES encourages Nosana to continue in this direction, as testing is the first step to security. Moreover, integration tests were useful to audit the security of the staking program.

Vulnerabilities summary

Total vulnerabilities	6
🟥 Critical	0
🟧 Major	0
🟨 Medium	1
🟩 Minor	2
🟦 Informational	3

Vulnerabilities & issues table

Identified vulnerabilities

Ref	Vulnerability title	Severity	Remediation effort
#1	Centralized slashing privileges	🟨 Medium	🟧 High
#2	Funds centralization	🟩 Minor	🟨 Medium
#3	Unsafe arithmetic	🟩 Minor	🟩 Low
#4	Misleading account names	🟦 Informative	🟩 Low
#5	Missing *stats* account validation on slash instruction	🟦 Informative	🟩 Low
#6	Use of Sysvar clock account	🟦 Informative	🟩 Low

Identified vulnerabilities

Centralized slashing privileges

Severity	Remediation effort
🟨 Medium	🟧 High

Description

The staking program implements a *slash* instruction allowing the *authority* account to withdraw tokens from a given user stake account.

programs/nosana-staking/src/instructions/slah.rs (L5)

```
#[derive(Accounts)]
pub struct Slash<'info> {
    #[account(mut)]
    pub ata_to: Box<Account<'info, TokenAccount>>,
    #[account(mut, seeds = [ nos::ID.key().as_ref() ], bump)]
    pub ata_vault: Box<Account<'info, TokenAccount>>,
    #[account(mut)]
    pub stake: Account<'info, StakeAccount>,
    #[account(mut, has_one = authority)]
    pub stats: Box<Account<'info, StatsAccount>>,
    pub authority: Signer<'info>,
    pub token_program: Program<'info, Token>,
}
```

Scope

Nosana staking program

Risk

The slashing privilege is overly centralized. A malicious administrator could withdraw the total liquidity of the staking program to any arbitrary token account.

This issue makes even more sense with the recent event regarding Slope wallets hack. It could be dangerous to have one wallet with total authority on the staking program.

Remediation

OPCODES engineers would recommend giving the slashing privileges to another program that would be responsible for slashing users based on a known set of rules.

Another possibility would be to give the slashing authority to a vote program or at least a multisig address.

Finally, OPCODES engineers think that the `ata_to` token account should not be an arbitrary token account. The slashed tokens should be either burned or sent to a known treasury address.

Funds centralization

Severity	Remediation effort
🟩 Minor	🟧 Medium

Description

When the staking program is initialized with the *init_vault* instruction, a new token account called *ata_vault* is created.

programs/nosana-staking/src/instructions/init_vault.rs (L5)

```rust
#[derive(Accounts)]
pub struct InitVault<'info> {
    [...]
    #[account(
        init,
        payer = authority,
        token::mint = mint,
        token::authority = ata_vault,
        seeds = [ mint.key().as_ref() ],
        bump,
    )]
    pub ata_vault: Box<Account<'info, TokenAccount>>,
    [...]
}
```

Upon staking, the program will always transfer the user's tokens to the same *ata_vault* token account. Meaning that all the staked funds are stored inside the same token account.

programs/nosana-staking/src/instructions/stake.rs (L29)

```rust
pub fn handler(ctx: Context<Stake>, amount: u64, duration: u64) -> Result<()> {
    [...]
    transfer_tokens(
        ctx.accounts.token_program.to_account_info(),
        ctx.accounts.ata_from.to_account_info(),
        ctx.accounts.ata_vault.to_account_info(),
        ctx.accounts.authority.to_account_info(),
        0, // skip signature
        amount,
    )?;
    [...]
```

Scope

Nosana staking program

Risk

At the time of writing, the global vault practice does not lead to any exploitable scenario. But from a security point of view, OPCODES would recommend segregating user funds when possible. It will make vulnerabilities affecting the user's funds harder to exploit.

Remediation

OPCODES engineers recommend creating a new vault account per user upon staking. Users will have to pay a rent for their token account which could be closed in the *claim* instruction (so users would get back their rent).

Unsafe arithmetic

Severity	Remediation effort
🟩 Minor	🟩 Low

Description

The program uses unsafe arithmetic operations that may lead to integer overflow/underflow.

programs/nosana-staking/src/state.rs (L34)

```
pub fn add(&mut self, amount: u128) {
    self.xnos += amount;
}

pub fn sub(&mut self, amount: u128) {
    self.xnos -= amount;
}
```

programs/nosana-staking/src/state.rs (L76)

```
pub fn topup(&mut self, amount: u64) {
    self.amount += amount;
    self.update_xnos();
}

pub fn slash(&mut self, amount: u64) {
    self.amount -= amount;
    self.update_xnos();
}

pub fn extend(&mut self, duration: u64) {
    self.duration += duration;
    self.update_xnos();
}
```

Scope

Nosana staking program

Risk

The risk is low as most of the unsafe operations are not exploitable because of proper input validation. The functions *add*, *sub*, *topup* and *slash*, are protected against overflows since their parameters can only vary in restrictive ranges. But in the case of the *extend* function, the *duration* parameter is a direct user input which can lead to an overflow. This overflow is harmless in the current version of the protocol but could potentially lead to vulnerabilities in the future.

Remediation

OPCODES engineers recommend to always use safe arithmetic, especially when user inputs are involved.

If you don't have any constraint on the compute units consumed, you can globally enable overflow/underflow checks by adding the following lines to the *cargo.toml* file.

```
[profile.release]
overflow-checks = true
```

Alternatively, if you prefer to use safe arithmetic on case-by-case basis you should use the following functions: *checked_mul*, *checked_div*, *checked_add* or *checked_sub*.

Misleading account names

Severity	Remediation effort
🟦 Informative	🟩 Low

Description

The staking program uses account names starting with *"ata"* to nominate regular token accounts. The ATA acronym stands for *Associated Token Account* and is meant to be used for token accounts created with the *Associated Token Program*.

programs/nosana-staking/src/instructions/init_vault.rs (L5)

```
#[derive(Accounts)]
pub struct InitVault<'info> {
    [...]
    #[account(
        init,
        payer = authority,
        token::mint = mint,
        token::authority = ata_vault,
        seeds = [ mint.key().as_ref() ],
        bump,
    )]
    pub ata_vault: Box<Account<'info, TokenAccount>>,
    [...]
}
```

Scope

Nosana staking program

Risk

The account names are misleading, and it makes the code harder to understand. Moreover, if you are planning to use ATAs, additional constraints should be implemented.

Remediation

OPCODES engineers recommend renaming account names to keep consistency between what the program is expecting and the variable names.

If you want to specifically use *Associated Token Account*, you should implement the constraints *associated_token::mint* and *associated_token::authority* as shown below.

```
#[account(
    associated_token::mint = mint,
    associated_token::authority = user,
)]
pub ata_user: Account<'info, TokenAccount>,
```

Missing *stats* account validation on *slash* instruction

Severity	Remediation effort
🟦 Informative	🟩 Low

Description

The staking program uses a global *stats* account in order to keep track of the amount of token stacked.

programs/nosana-staking/src/instructions/init_vault.rs (L5)

```
#[derive(Accounts)]
pub struct InitVault<'info> {
    #[account(init, payer = authority, space = STATS_SIZE, seeds = [ b"stats" ], bump)]
    pub stats: Box<Account<'info, StatsAccount>>,
    [...]
}
```

The *slash* and *update_authority* instructions are missing the seed constraint.

programs/nosana-staking/src/instructions/slash.rs (L5)

```
#[derive(Accounts)]
pub struct Slash<'info> {
    [...]
    #[account(mut, has_one = authority)]
    pub stats: Box<Account<'info, StatsAccount>>,
    [...]
}
```

Scope

Nosana staking program

Risk

As there can be only one *stats* account, this issue does not lead to any exploitable scenario. But in the future, if support for multiple pools is added, this issue could become a problem.

Remediation

From a defense in depth perspective, OPCODES engineers recommend validating the seed of every PDA that the program is using.

Use of Sysvar clock account

Severity	Remediation effort
🟦 Informative	🟩 Low

Description

The staking program is using the Sysvar clock account in order to query the timestamp of the transaction's slot.

programs/nosana-staking/src/instructions/stake.rs (L5)

```
#[derive(Accounts)]
pub struct Stake<'info> {
    [...]
    pub clock: Sysvar<'info, Clock>,
    pub system_program: Program<'info, System>,
    pub token_program: Program<'info, Token>,
}
```

Scope

Nosana staking program

Risk

Using Sysvar clock account is inefficient and increase the size of the transaction and the number of accounts used.

Remediation

OPCODES engineers recommend querying the clock at runtime using the *get()* function.

```
let clock = Clock::get()?;
let slot_time: i64 = clock.unix_timestamp;
```

Conclusion

OPCODES evaluated the security level of Nosana staking program as good. Nosana team was proactive in answering questions about the audit and the code logic. They demonstrated a good understanding of the Solana ecosystem and security hygiene. Indeed, Nosana codebase is well designed, each use case of their solution is segregated inside its own program. Therefore, a vulnerability in the reward program would not result in a loss of user funds in the staking program. Moreover, the staking program is correctly covered by integration tests, and both successful and unsuccessful scenarios are tested.

Nonetheless, the assessment demonstrated the presence of one medium vulnerability. It concerns the centralization of slashing privilege. This issue makes even more sense with the recent event regarding Slope wallets hack. OPCODES would recommend finding a way to handle the slashing more transparently with a proper set of rules and a known treasury address.

Finally, for an in-depth defense, OPCODES recommend fixing the minor issues regarding the segregation of user s' funds and the overflow in the extend function.