SECURITY AUDIT OF

NOXAL VESTING SMART CONTRACT

Public Report

Nov 29, 2022

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Solana	A decentralized blockchain built to enable scalable, user-friendly apps for the world.
SOL	A cryptocurrency whose blockchain is generated by the Solana platform.
Lamport	A fractional native token with the value of 0.000000001 sol.
Program	An app interacts with a Solana cluster by sending it transactions with one or more instructions. The Solana runtime passes those instructions to program.
Instruction	The smallest contiguous unit of execution logic in a program.
Cross-program invocation (CPI)	A call from one smart contract program to another.
Anchor	A framework for Solana's Sealevel runtime providing several convenient developer tools for writing smart contracts.
DAO	A digital Decentralized Autonomous Organization and a form of investor-directed venture capital fund.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Nov 29, 2022. We would like to thank the Noxal for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Noxal Vesting Smart Contract. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified no vulnerable issue in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Noxal Vesting Smart Contract

A world of alien-invading and robot-fighting epicness. 9,999 exclusive NFTs for #Solana War

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the Noxal Vesting Smart Contract. It was conducted on commit `00c2646e1e2e23205c55bc554c86ce6c5c49fb0f` from git repository link: *https://gitlab.com/t8526/noxel-vesting*.

The latest version of the following file was made available in the course of the review:

SHA256 Sum	File
`37e8b9b305c5dde93beaba3e9613ff508a55d1117fa2865bebd55a927e32bbf4`	`entrypoint.rs`
`6cda011e0517db456bf86beb44cd12d063078e558660c75351755e090ae885b5`	`error.rs`
`9e83b6a29563a347fce0bb6f6f78773f07bffad9e580c209f194ff86054c3bbe`	`instruction.rs`
`ddd41d939725dfb0dd8fe7c42993a3ad1bcd95d06d0af5fd3cb97384740cbf55`	`lib.rs`
`80bddd0e8de0c71126db9705be9c44eda66445e01ac98adfac95daa152e9baaa`	`processor.rs`
`b4d66f35f007fb2de9206898cdf6c4ca0b59a3f64a22264794347f5c80fc6786`	`state.rs`

1.3. Audit methodology

Our security audit process for Solana smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the Solana smart contract:

- Arithmetic Overflow and Underflow
- Signer checks
- Ownership checks
- Rent exemption checks
- Account confusions

- Bump seed canonicalization
- Closing account
- Signed invocation of unverified programs
- Numerical precision errors
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The Noxal Vesting Smart Contract was written in `Rust` programming language.

It's a vesting contract which allows anyone to deposit SPL tokens that are unlocked to a specified public key at a certain block height/slot. The recipient address can be modified by the owner of the current recipient key.

2.2. Findings

During the audit process, the audit team found no vulnerability in the given version of Noxal Vesting Smart Contract, only some notes and recommendations.

2.2.1. processor.rs - Should get `timestamp` by using `Clock::get()` instead INFORMATIVE

The contract should get `timestamp` by using `Clock::get()` instead because it is easier, more efficient and does not require that the `clock_sysvar_account` account be passed to the program, or specified in the `Instruction` the program is processing.

```
pub fn process_unlock(
    program_id: &Pubkey,
    _accounts: &[AccountInfo],
    seeds: [u8; 32],
) -> ProgramResult {
    ...
    // Unlock the schedules that have reached maturity
    let clock = Clock::from_account_info(&clock_sysvar_account)?;
    let mut total_amount_to_transfer = 0;
    let mut schedules =
unpack_schedules(&packed_state.borrow()[VestingScheduleHeader::LEN..])?;

    for s in schedules.iter_mut() {
        if clock.unix_timestamp as u64 >= s.release_time {
            total_amount_to_transfer += s.amount;
            s.amount = 0;
        }
    }
    ...
}
```

RECOMMENDATION

Using `Clock::get()` instead of `Clock::from_account_info(&clock_sysvar_account)`.

```
pub fn process_unlock(
    program_id: &Pubkey,
    _accounts: &[AccountInfo],
    seeds: [u8; 32],
) -> ProgramResult {
    ...
    // Unlock the schedules that have reached maturity
    let clock = Clock::get();
    let mut total_amount_to_transfer = 0;
    let mut schedules =
unpack_schedules(&packed_state.borrow()[VestingScheduleHeader::LEN..])?;

    for s in schedules.iter_mut() {
        if clock.unix_timestamp as u64 >= s.release_time {
            total_amount_to_transfer += s.amount;
            s.amount = 0;
        }
    }
    ...
}
```

2.2.2. processor.rs - Unnecessary SPL program ID check INFORMATIVE

The `spl-token` program from version `v3.1.1` includes `check_program_account` in all the token instruction functions to ensure the user-provided `token_program_id` is the same as the spl-token `program_id`. The current contract is using `spl-token v3.5.0` so it is unnecessary to perform this check.

```
pub fn process_unlock(
    program_id: &Pubkey,
    _accounts: &[AccountInfo],
    seeds: [u8; 32],
) -> ProgramResult {
    let accounts_iter = &mut _accounts.iter();

    let spl_token_account = next_account_info(accounts_iter)?;
    let clock_sysvar_account = next_account_info(accounts_iter)?;
    let vesting_account = next_account_info(accounts_iter)?;
    let vesting_token_account = next_account_info(accounts_iter)?;
    let destination_token_account = next_account_info(accounts_iter)?;

    let vesting_account_key = Pubkey::create_program_address(&[&seeds], program_id)?;
    if vesting_account_key != *vesting_account.key {
        msg!("Invalid vesting account key");
        return Err(ProgramError::InvalidArgument);
    }

    // Unnecessary check
    if spl_token_account.key != &spl_token::id() {
        msg!("The provided spl token program account is invalid");
        return Err(ProgramError::InvalidArgument)
    }
    ...
}

// spl-token 3.5.0
solana_program::declare_id!("TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA");

pub fn check_program_account(spl_token_program_id: &Pubkey) -> ProgramResult {
    if spl_token_program_id != &id() {
        return Err(ProgramError::IncorrectProgramId);
    }
    Ok(())
}
```

RECOMMENDATION

Removing unnecessary check.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Nov 29, 2022*	Public Report	Verichains Lab

Table 2. Report versions history