Security Audit Report for Allstake Solana Program

Date: July 5, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Allstake
Target	Allstake Solana Program

Version History

Version	Date	Description
1.0	July 5, 2024	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-07-05 11:35:53+0000

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Rust
Approach	Semi-automatic and manual verification

The focus of this audit is the `programs/strategy-manager` within the Allstake Solana Program [1]. Please note that other external dependencies in the repository, including the solana development framework `Anchor` [2], are considered reliable in terms of both functionality and security, these files are not included in the scope of the audit.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Allstake Solana Program	`Version 1`	`0ee2d60b3e52a8389a9e3b84624d30dae9b68813`
	`Version 2`	`6c35aae795a7e85a0eb75ce80beb82cad43c52f4`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

[1] `https://github.com/allstake/allstake-solana-program`

[2] `https://www.anchor-lang.com/`

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

∗ Reentrancy
∗ DoS
∗ Access control
∗ Data handling and data flow
∗ Exception handling
∗ Untrusted external call and control flow
∗ Initialization consistency
∗ Events operation
∗ Error-prone randomness
∗ Improper use of the proxy system

1.3.2 DeFi Security

∗ Semantic consistency
∗ Functionality consistency
∗ Permission management
∗ Business logic
∗ Token operation
∗ Emergency mechanism
∗ Oracle security
∗ Whitelist and blacklist
∗ Economic impact
∗ Batch transfer

1.3.3 NFT Security

∗ Duplicated item
∗ Verification of the token receiver
∗ Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [3] and Common Weakness Enumeration [4]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ **Likelihood**	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[4] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **three** potential security issues. Besides, we have **one** recommendation and **one** note.

- High Risk: 2
- Low Risk: 1
- Recommendation: 1
- Note: 1

ID	Severity	Description	Category	Status
1	High	Potential DoS in function `create_strategy()`	Software Security	Fixed
2	High	Pool drained by depositing fake tokens	DeFi Security	Fixed
3	Low	Potential delay of withdrawal in function `queue_withdraw()`	DeFi Security	Confirmed
4	-	Lack of check for updating system variables	Recommendation	Fixed
5	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential DoS in function `create_strategy()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `create_strategy()`, when creating a strategy corresponding to a specific `strategy_mint`, an `associated token account` (ATA) will be create accordingly. Here, the Anchor's `init` implementation is used, which will check whether the account (i.e., `strategy_ata`) has already been created. If it has been created, the execution will revert. However, the `associated token program` allows users to create corresponding ATAs for other users. If the ATAs are created in advance, the function `create_strategy()` can be vulnerable to a DoS attack.

```
#[derive(Accounts)]
pub struct CreateStrategy<'info> {
    #[account(
        mut,
        seeds = [
            b"STATE",
        ],
        bump,
    )]
    pub state: Box<Account<'info, StrategyManager>>,
    #[account(
        init,
        payer = signer,
        space = Strategy::serialized_len(),
        seeds = [
            b"STRATEGY",
            strategy_mint.key().as_ref(),
        ],
        bump,
    )]
    pub strategy: Box<Account<'info, Strategy>>,
    #[account(
        init,
        payer = signer,
        associated_token::mint = strategy_mint,
        associated_token::authority = strategy,
    )]
    pub strategy_ata: Account<'info, TokenAccount>,
    pub strategy_mint: Account<'info, Mint>,
    #[account(mut)]
    pub signer: Signer<'info>,
    pub system_program: Program<'info, System>,
    pub token_program: Program<'info, Token>,
    pub associated_token_program: Program<'info, AssociatedToken>,
}
```

Listing 2.1: programs/strategy-manager/src/instructions.rs

Impact The pre-creation of ATAs for the secondary vault can lead to a potential DoS vulnerability in the `create_strategy()` function.

Suggestion Use `init_if_needed` instead of `init` for `strategy_ata`.

2.2 DeFi Security

2.2.1 Pool drained by depositing fake tokens

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `deposit()`, `from_ata` and `strategy_ata` are supposed to represent the `associated token account` of the user and strategy respectively. However, there is no validation to ensure these `associated token accounts` passed definitely correspond to the strategy required.

In this case, the attacker can pass in two `associated token accounts` corresponding to fake tokens to execute the transfer logic, while the function still considers that the valuable tokens corresponding to the strategy have been received, and updates the `user_deposit` and strategy accounts for further withdrawal.

```rust
#[derive(Accounts)]
pub struct Deposit<'info> {
    #[account(mut)]
    pub from: Signer<'info>,
    #[account(mut)]
    pub from_ata: Account<'info, TokenAccount>,
    #[account(mut)]
    pub strategy_ata: Account<'info, TokenAccount>,
    #[account(
        mut,
        seeds = [
            b"DEPOSIT",
            from.key().as_ref(),
            strategy_mint.key().as_ref()
        ],
        bump,
    )]
    pub user_deposit: Account<'info, UserDeposit>,
    #[account(
        mut,
        seeds = [
            b"STRATEGY",
            strategy_mint.key().as_ref(),
        ],
        bump
    )]
    pub strategy: Account<'info, Strategy>,
    pub strategy_mint: Account<'info, Mint>,
    pub token_program: Program<'info, Token>,
    pub system_program: Program<'info, System>,
}

impl<'info> Deposit<'info> {
    pub fn process(&mut self, amount: u64) -> Result<()> {
        let mut strategy: StrategyV1 = self.strategy.data.into();
        let mut user_deposit: UserDepositV1 = self.user_deposit.data.into();

        if strategy.deposit_paused {
            return err!(StrategyManagerError::DepositPaused);
        }

        // transfer tokens to strategy ATA
        let cpi_accounts = SplTransfer {
            from: self.from_ata.to_account_info(),
            to: self.strategy_ata.to_account_info(),
            authority: self.from.to_account_info(),
        };
        token::transfer(
            CpiContext::new(self.token_program.to_account_info(), cpi_accounts),
            amount,
        )?;

        // update user deposit and strategy state
        user_deposit.deposited = user_deposit.deposited.checked_add(amount).unwrap();
        strategy.total_deposited = strategy.total_deposited.checked_add(amount).unwrap();

        self.user_deposit.data = user_deposit.into();
        self.strategy.data = strategy.into();

        emit!(UserDepositEvent {
            user: self.from.key(),
            mint: self.strategy_mint.key(),
            amount,
            strategy_id: strategy.id,
        });

        Ok(())
    }
}
```

Listing 2.2: programs/strategy-manager/src/instructions/deposit.rs

Impact Pools can be drained by depositing fake tokens.

Suggestion Add checks to ensure `strategy_ata`'s authority is strategy, and its mint is `strategy_mint`.

2.2.2 Potential delay of withdrawal in function `queue_withdraw()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description According to the design, the user's withdrawal is divided into two steps: `queue_withdraw()` and `complete_withdraw()`. When the user invokes function `queue_withdraw()`, it will record a withdrawal start time as `withdraw_started_at`. The user can only invoke the function `complete_withdraw()` to take out the token after the specified `withdraw_delay` period has passed. During this period or after it has passed without invoking `complete_withdraw()`, if the user invokes `queue_withdraw()` again to make an additional withdrawal, the withdrawal start time recorded will be reset to the current time, effectively delaying the time at which the user can complete the withdrawal.

```
impl<'info> QueueWithdraw<'info> {
    pub fn process(&mut self, amount: u64) -> Result<()> {
        let mut user_deposit: UserDepositV1 = self.user_deposit.data.into();
        let mut strategy: StrategyV1 = self.strategy.data.into();

        if strategy.withdraw_paused {
            return err!(StrategyManagerError::WithdrawPaused);
        }

        if user_deposit.deposited < amount {
            return err!(StrategyManagerError::InsufficientBalance);
        }

        // update states
        user_deposit.deposited = user_deposit.deposited.checked_sub(amount).unwrap();
        strategy.total_deposited = strategy.total_deposited.checked_sub(amount).unwrap();

        user_deposit.queued_withdraw_amount = user_deposit
            .queued_withdraw_amount
            .checked_add(amount)
            .unwrap();
        let now_ts = Clock::get().unwrap().unix_timestamp;
        user_deposit.withdraw_started_at = now_ts;

        self.user_deposit.data = user_deposit.into();
        self.strategy.data = strategy.into();

        emit!(events::QueueWithdrawEvent {
            user: self.receiver.key(),
            mint: self.strategy_mint.key(),
            amount,
            strategy_id: strategy.id,
        });

        Ok(())
    }
}
```

Listing 2.3: programs/strategy-manager/src/instructions/queue_withdraw.rs

Impact The user's withdrawal can be delayed.

Suggestion For each withdrawal request, recording a start time.

Feedback from the Project This is an expected behavior for now, many staking protocols have similar process. In the future, we may update it to have a better user experience.

2.3 Additional Recommendation

2.3.1 Lack of check for updating system variables

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The privileged owner can pause/unpause deposits and withdrawals from being processed for a specific strategy via the function `pause_strategy()`. However, there is no check whether the strategy is already paused/unpaused when updating.

Similar problems also exist in functions `set_strategy_withdraw_delay()`, `set_min_withdraw_delay()`, and `set_owner()`.

```
impl<'info> PauseStrategy<'info> {
    pub fn process(&mut self, pause_deposit: bool, pause_withdraw: bool) -> Result<()> {
        let state: StrategyManagerV1 = self.state.data.into();
        if state.owner.key() != *self.signer.key {
            return err!(StrategyManagerError::NotOwner);
        }

        let mut strategy: StrategyV1 = self.strategy.data.into();
        strategy.deposit_paused = pause_deposit;
        strategy.withdraw_paused = pause_withdraw;
        self.strategy.data = strategy.into();

        emit!(events::PauseStrategy {
            strategy_mint: self.strategy_mint.key(),
            strategy_id: strategy.id,
            deposit_paused: pause_deposit,
            withdraw_paused: pause_withdraw,
        });

        Ok(())
    }
}
```

Listing 2.4: programs/strategy-manager/src/instructions/pause_strategy.rs

Suggestion Add checks to ensure new configurations are different from the origins before updating.

2.4 Note

2.4.1 Potential centralization risks

Introduced by `Version 1`

Description The owner has the ability to upgrade the program account. Besides, the owner role of the state account can pause/unpause deposits and withdrawals from being processed for a specific strategy and update global system variables (i.e., `withdraw_delay` and `min_withdraw_delay`).

Feedback from the Project We will transfer the ownership to DAO soon after the launch on solana.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS