BLOCKSEC

Security Audit

Report for restaking program

Date: June 26, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction **1**

1.1 About Target Contracts . . . 1
1.2 Disclaimer . . . 1
1.3 Procedure of Auditing . . . 2
1.3.1 Software Security . . . 2
1.3.2 DeFi Security . . . 2
1.3.3 NFT Security . . . 2
1.3.4 Additional Recommendation . . . 3
1.4 Security Model . . . 3

Chapter 2 Findings **4**

2.1 DeFi Security . . . 4
2.1.1 Incorrect price check for `WSOL` . . . 4
2.1.2 Lack of access control in privileged functions . . . 6
2.1.3 Incorrect minimum value check in `unstake()` . . . 8
2.1.4 Lack of configuration for `token_deposit_cap` . . . 8
2.1.5 Potential DoS in function `create_secondary_vault()` . . . 9
2.2 Note . . . 10
2.2.1 Timely update price before unstaking . . . 11
2.2.2 Insufficient price update before staking . . . 11
2.2.3 Potential out-of date prices for liquid staking tokens . . . 12
2.2.4 Security of related liquid staking protocols and tokens . . . 13
2.2.5 Potential centralization risks . . . 13

Report Manifest

Item	Description
Client	MetaPool
Target	restaking program

Version History

Version	Date	Description
1.0	June 26, 2024	First release

Signature



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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-06-27 02:27:43+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

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/mp-sol-restaking` within the restaking 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
restaking program	`Version 1`	`f73e802f90b10a95a82723721809ca5b4f673871`
	`Version 2`	`a2c5c3d9a95c772d9bf89ff9a725fde941ae53b7`

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/Meta-Pool/solana-restaking/tree/main/programs/mp-sol-restaking

[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 **five** potential security issues. Besides, we have **five** notes.

- High Risk: 3
- Medium Risk: 1
- Low Risk: 1
- Note: 5

ID	Severity	Description	Category	Status
1	High	Incorrect price check for `WSOL`	DeFi Security	Fixed
2	High	Lack of access control in privileged functions	DeFi Security	Fixed
3	Low	Incorrect minimum value check in `unstake()`	DeFi Security	Fixed
4	Medium	Lack of configuration for `token_deposit_cap`	DeFi Security	Fixed
5	High	Potential DoS in function `create_secondary_vault()`	DeFi Security	Fixed
6	-	Timely update price before unstaking	Note	-
7	-	Insufficient price update before staking	Note	-
8	-	Potential out-of date prices for liquid staking tokens	Note	-
9	-	Security of related liquid staking protocols and tokens	Note	-
10	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Incorrect price check for `WSOL`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `stake` serves for staking `LST` tokens and issuing `mpSOL` as a receipt token. The function first updates the `LST/SOL` price for a specific `secondary vault` and then uses that price to calculate the amount of `mpSOL` to be minted. There exists a requirement that the `vault_state.lst_sol_price_p32 > TWO_POW_32`. However, in the function `internal_update_vault_token_sol_price`, the `lst_sol_price_p32` for a `WSOL secondary vault` is always 1 (i.e., `TWO_POW_32`). This conflicts with the price constraint in the function `stake`, leading to failures when users attempt to stake into a `WSOL secondary vault`.

```
69    pub fn handle_stake(ctx: Context<Stake>, lst_amount: u64) -> Result<()> {
70        // check deposits are enabled in this secondary-vault
71        require_eq!(
```

```
72          ctx.accounts.vault_state.deposits_disabled,
73          false,
74          ErrorCode::DepositsInThisVaultAreDisabled
75      );
76
77      // check token_sol_price is in range and not stale
78      // LST/SOL price must be > 1
79      require_gt!(
80          ctx.accounts.vault_state.lst_sol_price_p32,
81          TWO_POW_32,
82          ErrorCode::InvalidStoredLstPrice
83      );
84      ...
```

Listing 2.1: src/instructions/users/stake.rs

```
58  pub fn internal_update_vault_token_sol_price(
59      main_state: &mut Account<MainVaultState>,
60      secondary_state: &mut Account<SecondaryVaultState>,
61      lst_state: Option<AccountInfo>
62  ) -> Result<()> {
63      //
64      let old_price_p32 = secondary_state.lst_sol_price_p32;
65
66      let new_price_p32 = match secondary_state.lst_mint.key() {
67          // wSol is simple, always 1
68          WSOL_MINT => TWO_POW_32,
69          ...
123     };
124
125     // only if price changed
126     if new_price_p32 != old_price_p32 {
127         ...
146         // update last price and timestamp
147         secondary_state.lst_sol_price_p32 = new_price_p32;
148         secondary_state.lst_sol_price_timestamp = Clock::get().unwrap().unix_timestamp as u64;
149
150         emit!(crate::events::UpdateVaultTokenSolPriceEvent {
151             main_state: main_state.key(),
152             lst_mint: secondary_state.lst_mint.key(),
153             lst_amount,
154             old_price_p32,
155             old_sol_value,
156             new_price_p32,
157             new_sol_value,
158             main_vault_backing_sol_value: main_state.backing_sol_value,
159         });
160     }
161     Ok(())
162 }
```

Listing 2.2: src/instructions/cranks/update_vault_sol_price.rs

Impact Users fail to stake into a `WSOL secondary vault`.

Suggestion Revise the price check in the function `stake`.

2.1.2 Lack of access control in privileged functions

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the current implementation, any user can create a new `main vault`, a `mpSOL mint`, and a `secondary vault`, and perform normal staking and unstaking operations. In this case, the `ticket account` generated from unstaking using these "fake" accounts can be used to claim in the officially created vaults for corresponding liquid staking tokens, leading to inconsistencies between the vaults' data and the actual state, resulting in a DoS.

```
30  pub fn handle_unstake(ctx: Context<Unstake>, mpsol_amount: u64) -> Result<()> {
31      // check mpsol_amount > MIN_MOVEMENT_LAMPORTS
32      require_gte!(
33          mpsol_amount,
34          MIN_MOVEMENT_LAMPORTS,
35          ErrorCode::UnstakeAmountToSmall
36      );
37
38      // compute the sol value of the mpsol to burn
39      let ticket_sol_value = mpsol_amount_to_sol_value(
40          mpsol_amount,
41          ctx.accounts.main_state.backing_sol_value,
42          ctx.accounts.mpsol_mint.supply,
43      );
44
45      // -------
46      // burn the mpSOL and keep contract internal accounting
47      // -------
48      burn(
49          CpiContext::new(
50              ctx.accounts.token_program.to_account_info(),
51              Burn {
52                  mint: ctx.accounts.mpsol_mint.to_account_info(),
53                  from: ctx.accounts.unstaker_mpsol_account.to_account_info(),
54                  authority: ctx.accounts.unstaker.to_account_info(),
55              },
56          ),
57          mpsol_amount,
58      )?;
59      // by removing from main_state.backing_sol_value,
60      // mpSOL price remains the same after the burn
61      ctx.accounts.main_state.backing_sol_value -= ticket_sol_value;
62      // -------
63
64      // compute ticket due timestamp
65      let now_ts = Clock::get().unwrap().unix_timestamp as u64;
```

```
66      let ticket_due_timestamp =
67          now_ts + ( ctx.accounts.main_state.unstake_ticket_waiting_hours as u64 * 60 * 60 );
68
69      // initialize new_ticket_account
70      ctx.accounts
71          .new_ticket_account
72          .set_inner(crate::state::UnstakeTicket {
73              main_state: ctx.accounts.main_state.key(),
74              beneficiary: ctx.accounts.unstaker.key(),
75              ticket_sol_value,
76              ticket_due_timestamp,
77          });
78      ...
```

Listing 2.3: src/instructions/users/unstake.rs

```
60  pub fn handle_ticket_claim(ctx: Context<TicketClaim>, withdraw_sol_value_amount: u64) ->
        Result<()> {
61      ...
138     // send tokens to the user
139     {
140         anchor_spl::token::transfer(
141         CpiContext::new_with_signer(
142             ctx.accounts.token_program.to_account_info(),
143             Transfer {
144                 from: ctx.accounts.vault_lst_account.to_account_info(),
145                 to: ctx.accounts.beneficiary_lst_account.to_account_info(),
146                 authority: ctx.accounts.vaults_ata_pda_auth.to_account_info(),
147             },
148             &[&[
149                 &ctx.accounts.main_state.key().to_bytes(),
150                 VAULTS_ATA_AUTH_SEED,
151                 &[ctx.bumps.vaults_ata_pda_auth]
152                 ]]
153         ),
154         lst_amount_to_deliver)?;
155     }
156     // the tokens are removed from the vault total
157     ctx.accounts.vault_state.vault_total_lst_amount -= lst_amount_to_deliver;
158     // and computed as locally stored amount
159     ctx.accounts.vault_state.locally_stored_amount -= lst_amount_to_deliver;
160
161     emit!(crate::events::TicketClaimEvent {
162         main_state: ctx.accounts.main_state.key(),
163         lst_mint: ctx.accounts.lst_mint.key(),
164         ticket_account: ctx.accounts.ticket_account.key(),
165         beneficiary: ctx.accounts.beneficiary.key(),
166         claimed_sol_value: withdraw_sol_value_amount,
167         ticket_sol_value_remaining: ctx.accounts.ticket_account.ticket_sol_value,
168         lst_amount_delivered: lst_amount_to_deliver,
169         ticket_due_timestamp: ctx.accounts.ticket_account.ticket_due_timestamp,
170     });
171
```

```
172         Ok(())
173     }
```

Listing 2.4: src/instructions/users/ticket_claim.rs

Impact The unstaking operation can suffer a DoS in the above scenario.

Suggestion Add access control to the related privileged functions.

2.1.3 Incorrect minimum value check in `unstake()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There is a check to ensure the minimum amount of `SOL` to be moved (i.e., `MIN_MOVEMENT_LAMPORTS`). However, in function `handle_unstake()`, the check is used in the validation of the amount of `mpSOL` to be burned, which is inconsistent with the design.

```
30  pub fn handle_unstake(ctx: Context<Unstake>, mpsol_amount: u64) -> Result<()> {
31      // check mpsol_amount > MIN_MOVEMENT_LAMPORTS
32      require_gte!(
33          mpsol_amount,
34          MIN_MOVEMENT_LAMPORTS,
35          ErrorCode::UnstakeAmountToSmall
36      );
37
38      // compute the sol value of the mpsol to burn
39      let ticket_sol_value = mpsol_amount_to_sol_value(
40          mpsol_amount,
41          ctx.accounts.main_state.backing_sol_value,
42          ctx.accounts.mpsol_mint.supply,
43      );
44      ...
```

Listing 2.5: src/instructions/users/unstake.rs

Impact The user has to burn more `mpSOLs` than expected.

Suggestion Check that the value of `SOL` corresponding to `mpSOL` is greater than the minimum value, rather than checking the amount of `mpSOL` itself.

2.1.4 Lack of configuration for `token_deposit_cap`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The parameters `token_deposit_cap` in the `SecondaryVaultState` represents the maximum capacity for a `secondary vault`, where a value of `ZERO` indicates no cap. Meanwhile, the function `stake` invokes the `SecondaryVault.check_cap` function to verify if it has reached this limitation. However, this parameter can only be initialized to be `ZERO` when creating the `secondary vault`, and there is no method available to modify it after the vault creation.

```rust
pub fn handle_create_secondary_vault(ctx: Context<CreateSecondaryVault>) -> Result<()> {
    ctx.accounts.vault_state.set_inner(SecondaryVaultState {
        lst_mint: ctx.accounts.lst_mint.key(),
        vault_total_lst_amount: 0,
        lst_sol_price_p32 : 0,
        lst_sol_price_timestamp : 0,
        in_strategies_amount : 0,
        locally_stored_amount : 0,
        tickets_target_sol_amount : 0,
        deposits_disabled : true,
        token_deposit_cap : 0,
    });
    Ok(())
}
```

Listing 2.6: src/instructions/admin/create_secondary_vault.rs

```rust
pub fn check_cap(&self) -> Result<()> {
    if self.token_deposit_cap > 0 {
        require_gte!(
            self.token_deposit_cap,
            self.locally_stored_amount + self.in_strategies_amount,
            ErrorCode::DepositExceedsVaultCap
        );
    }
    Ok(())
}
```

Listing 2.7: src/state/secondary_vault.rs

Impact The cap check is invalid due to the absence of methods for updating the `token_deposit_cap` parameter.

Suggestion Implement the updating methods for key configuration, such as the `token_deposit_cap` parameter for a `secondary vault`.

2.1.5 Potential DoS in function `create_secondary_vault()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `create_secondary_vault()`, when creating a `secondary vault` corresponding to a specific `LST`, the protocol will create the `associated token accounts` (ATAs) to receive the corresponding `LST`. The Anchor's `init` constraint is used, which checks and reverts if the account has already been created. However, the `associated token program` allows users to create corresponding ATAs for other users. A malicious actor can DoS the function `create_secondary_vault()` through creating specific ATAs in advance.

```rust
#[derive(Accounts)]
pub struct CreateSecondaryVault<'info> {
```

```
16      #[account(mut)]
17      pub admin: Signer<'info>,
18
19      #[account(mut, has_one=admin)]
20      pub main_state: Account<'info, MainVaultState>,
21
22      #[account(mint::decimals = 9)]
23      // all LST mints must have 9 decimals, to simplify x/SOL price calculations
24      pub lst_mint: Account<'info, Mint>,
25
26      // secondary vaults are PDAs of main_state
27      // only this program & main_state can create a secondary vault
28      #[account(init, payer = admin, space = 8 + SecondaryVaultState::INIT_SPACE,
29          seeds = [
30              &main_state.key().to_bytes(),
31              &lst_mint.key().to_bytes(),
32          ],
33          bump
34      )]
35      pub vault_state: Account<'info, SecondaryVaultState>,
36
37      /// CHECK: Auth PDA
38      #[account(
39          seeds = [
40              &main_state.key().to_bytes(),
41              VAULTS_ATA_AUTH_SEED
42          ],
43          bump
44      )]
45      pub vaults_ata_pda_auth: UncheckedAccount<'info>,
46
47      #[account(init, payer = admin,
48          associated_token::mint = lst_mint,
49          associated_token::authority = vaults_ata_pda_auth
50      )]
51      pub vault_lst_account: Account<'info, TokenAccount>,
52
53      pub associated_token_program: Program<'info, AssociatedToken>,
54      pub token_program: Program<'info, Token>,
55      pub system_program: Program<'info, System>,
56  }
```

Listing 2.8: src/instructions/admin/create_secondary_vault.rs

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

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

2.2 Note

2.2.1 Timely update price before unstaking

Introduced by Version 1

Description In the current implementation, users must first invoke the function `unstake` to generate a ticket and then withdraw their assets with the `SOL value` recorded in the ticket. However, the `SOL value` in the ticket has been calculated in the function `unstake` and based on the last updated `backing_sol_value` in the `main vault` instead of updating the vault status and including accumulated profits and loss first. If the status is not timely synchronized, users may potentially lose some rewards when they unstake.

```
30  pub fn handle_unstake(ctx: Context<Unstake>, mpsol_amount: u64) -> Result<()> {
31      // check mpsol_amount > MIN_MOVEMENT_LAMPORTS
32      require_gte!(
33          mpsol_amount,
34          MIN_MOVEMENT_LAMPORTS,
35          ErrorCode::UnstakeAmountToSmall
36      );
37
38      // compute the sol value of the mpsol to burn
39      let ticket_sol_value = mpsol_amount_to_sol_value(
40          mpsol_amount,
41          ctx.accounts.main_state.backing_sol_value,
42          ctx.accounts.mpsol_mint.supply,
43      );
44
45      // -------
46      // burn the mpSOL and keep contract internal accounting
47      // -------
48      burn(
49          CpiContext::new(
50              ctx.accounts.token_program.to_account_info(),
51              Burn {
52                  mint: ctx.accounts.mpsol_mint.to_account_info(),
53                  from: ctx.accounts.unstaker_mpsol_account.to_account_info(),
54                  authority: ctx.accounts.unstaker.to_account_info(),
55              },
56          ),
57          mpsol_amount,
58      )?;
59      ...
```

Listing 2.9: src/instructions/users/unstake.rs

Feedback from the Project It is designed on purpose as a form of "unstake fee" that benefits the rest of the mpSOL holders. The described behavior is a design decision, not a bug. When a user unstakes, they get the SOL-value computed at the moment of ticket creation, but they have to wait for withdrawal. The rewards difference benefits all other mpSOL holders.

2.2.2 Insufficient price update before staking

Introduced by Version 1

Description In the staking operation, the protocol will update the `LST` price corresponding to the specified `secondary vault`, and then update the `backing_sol_value` in the `main state` based on the price. However, this update of the `backing_sol_value` is not sufficient, as it only updates the `LST` from a single `secondary vault` instead of all the `secondary vaults` (including the revenue generated by subsequent strategies). Since the `backing_sol_value` determines the price of `mpSOL`, the incomplete update prior to staking may result in a lower `mpSOL` price, allowing stakers to mint more shares.

```
69  pub fn handle_stake(ctx: Context<Stake>, lst_amount: u64) -> Result<()> {
70      // check deposits are enabled in this secondary-vault
71      require_eq!(
72          ctx.accounts.vault_state.deposits_disabled,
73          false,
74          ErrorCode::DepositsInThisVaultAreDisabled
75      );
76
77      // check token_sol_price is in range and not stale
78      // LST/SOL price must be > 1
79      require_gt!(
80          ctx.accounts.vault_state.lst_sol_price_p32,
81          TWO_POW_32,
82          ErrorCode::InvalidStoredLstPrice
83      );
84
85      // we need the LST/SOL price to be updated
86      // update LST/SOL price now
87      internal_update_vault_token_sol_price(
88          &mut ctx.accounts.main_state,
89          &mut ctx.accounts.vault_state,
90          if ctx.remaining_accounts.len() >= 1 {Some(ctx.remaining_accounts[0].to_account_info())
                } else {None})?;
91      ...
```

Listing 2.10: src/instructions/users/stake.rs

Feedback from the Project It is working as designed. The price update on the secondary-vault is made only to compute the SOL value of the deposited LST. It is not expected that the smart contract executes recursive secondary-vault & strategies SOL-value updates on each stake.

The complete update of the total SOL-value with the corresponding change in mpSOL/SOL price is executed in parts, periodically, and by cranks. It is expected that staking before one of these partial updates provides a lower mpSOL/SOL price.

2.2.3 Potential out-of date prices for liquid staking tokens

Introduced by `Version 1`

Description The function `internal_update_vault_sol_price` fetches the `lst_sol_price` from external sources, such as `Marinade`. However, these external sources may not always provide timely updates. As a result, the prices used for `lst_sol_price` calculation may become outdated.

Feedback from the Project The only true price for mSOL is in Marinade's contract. That's the price at which unstake happens in Marinade's contract, meaning that's the price that the Marinade contract uses to deliver SOL in exhange for mSOL, meaning that the Marinade contract is the only authoritative source for mSOL value.

2.2.4 Security of related liquid staking protocols and tokens

Introduced by `Version 1`

Description The protocol uses liquid staking tokens from various liquid staking protocols as the collateral (e.g., `WSOL`, `MSOL`, and `SPL`). During the audit process, we assume that these protocols are secure, and the value of the corresponding tokens against `SOL` cannot be maliciously manipulated.

Feedback from the Project All sources have multiple audits, are blue-chips LST and have been operating for years without issues.

2.2.5 Potential centralization risks

Introduced by `Version 1`

Description The owner has the ability to upgrade the program account. Besides, the admin of the `main vault` and `secondary vault` can disable the staking functionality, update global system variables (e.g., `unstake_ticket_waiting_hours` and `performance_fee_bp`), and attach strategies to `secondary vaults`, which may result in centralization risks.

Feedback from the Project All those functions will be eventually derived to Meta Pool DAO for approval and execution. Initially it will be managed by a Multisig composed with the team and selected partners. This is the standard practice for Solana protocols.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS