BLOCKSEC

Security Audit
Report for Rhea LSD

Date: March 24, 2026 **Version:** 2.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Rhea Finance
Target	Rhea LSD

Version History

Version	Date	Description
1.0	January 22, 2026	First release
2.0	March 24, 2026	Remove wormhole

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-03-24 08:22:38+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 the Audit Target

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

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of Rhea LSD of Rhea Finance.

The protocol is deployed on the NEAR network and issues an LSD (Liquid Staking Derivative) token. Users deposit underlying tokens into the protocol, which are then supplied to Burrowland for yield generation. Privileged roles periodically withdraw rewards from Burrowland, swap them into underlying tokens, and re-deposit them to compound returns. Users can unstake at any time by burning their LSD tokens to redeem the corresponding underlying assets. The protocol also introduces cross-chain functionality, enabling users to complete both cross-chain transfers and deposits of underlying tokens within a single transaction.

Note this audit only focuses on the smart contracts in the following directories/files:

- contracts/rhea_lsd/src

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

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. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
Rhea LSD	`Version 1`	`2c4d8f83c78211bcd582bce7f47a37ed0912c317`
	`Version 2`	`63166484c16a5e32dcc8d85172ac825792e93988`
	`Version 3`	`e09858be2873d3e6f070edd739e3ff9f1c556467`

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.

[1] https://github.com/rhea-finance/rhea-lsd

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 Target, 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 the Target.

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.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target 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 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism

* Economic and incentive impact

1.3.2 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 [2] and Common Weakness Enumeration [3]. 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 five 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.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

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

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

Chapter 2 Findings

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

- High Risk: 1
- Medium Risk: 3
- Recommendation: 1
- Note: 3

ID	Severity	Description	Category	Status
1	High	Nonce derivation collision from block height dependency	Security Issue	Fixed
2	Medium	Lack of handling of cross-contract mint confirmation	Security Issue	Fixed
3	Medium	Funds locked due to lack of fallback mechanism in Wormhole `VAA` execution	Security Issue	Fixed
4	Medium	Unrecoverable reward assets from cross-contract transfer handling	Security Issue	Fixed
5	-	Add an executed `VAA` check in function `to_submit_vaa()`	Recommendation	Confirmed
6	-	Ensure timely invocation of function `rebalance()` to update the share value	Note	-
7	-	Potential centralization risks	Note	-
8	-	Assumption on `DEX` compatibility	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Nonce derivation collision from block height dependency

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `internal_sign_wormhole_transfer_call_tx()` generates an `MPC` signed `NEAR` transaction by constructing a `FunctionCall` action, requesting the `MPC` contract to sign it, and emitting the signed transaction data for an off-chain relayer to submit on-chain. The transaction nonce is produced by function `get_nonce()`, which derives the value solely from the current block height multiplied by a constant factor. As a result, multiple invocations of the function `internal_sign_wormhole_transfer_call_tx()` within the same block produce transactions with identical nonces. Since `NEAR` requires nonces to be strictly increasing per signer, only one of these transactions can be accepted, while all others are rejected due to nonce reuse.

```
160    pub fn internal_sign_wormhole_transfer_call_tx(
161        &self,
```

```
162         token_id: &AccountId,
163         amount: u128,
164         block_hash: Base58CryptoHash,
165         msg: &str,
166     ) -> Promise {
167         let signer_account_id = env::current_account_id();
168         let public_key = self.wormhole_state.sign_pubkey.clone().unwrap();
169         let function_call_action = FunctionCallAction {
170             method_name: "ft_transfer_call".to_string(),
171             args: json!({
172                 "receiver_id": self.wormhole_state.wormhole_bridge_id,
173                 "amount": U128(amount),
174                 "msg": msg,
175             })
176             .to_string()
177             .into_bytes(),
178             gas: Gas::from_tgas(300).as_gas(),
179             deposit: 1,
180         };
181         let tx = NearTransaction::V0(NearTransactionV0 {
182             signer_id: signer_account_id.clone(),
183             public_key: NearPublicKey::SECP256K1(Secp256K1PublicKey(
184                 public_key.as_bytes()[1..].try_into().unwrap(),
185             )),
186             nonce: get_nonce(),
187             receiver_id: token_id.clone(),
188             block_hash: CryptoHash(block_hash.into()),
189             actions: vec![Action::FunctionCall(Box::new(function_call_action))],
190         });
191         let tx_bytes = near_sdk::borsh::to_vec(&tx).unwrap();
192         let payload = env::sha256_array(&tx_bytes);
193         self.sign_near_tx_promise(
194             &signer_account_id,
195             hex::encode(tx_bytes),
196             Payload::Ecdsa(hex::encode(payload)),
197             0,
198         )
199     }
```

Listing 2.1: contracts/rhea_lsd/src/mpc.rs

```
34  pub fn get_nonce() -> u64 {
35      near_sdk::env::block_height() * 10u64.pow(6)
36  }
```

Listing 2.2: contracts/rhea_lsd/src/utils.rs

Impact Legitimate transactions may consistently fail when signed within the same block, causing denial of service for `Wormhole` related operations. At the same time, funds are stuck in the contract and can not be withdrawn.

Suggestion Revise the logic to ensure that the `nonce` in each transaction does not repeat.

2.1.2 Lack of handling of cross-contract mint confirmation

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Users enter the `LSD` system by transferring underlying tokens to the `LSD` contract, which then supplies these tokens into `Burrowland` to obtain corresponding shares and mints them to users. This process involves multiple cross-contract calls. After `Burrowland` receives the underlying tokens and converts them into shares internally, it calls back into the `LSD` contract through the function `on_burrowland_supply_client_echo()` to mint the corresponding shares for the user.

However, `Burrowland` does not process the result of this cross-contract call and directly returns `PromiseOrValue::Value(U128(0))`, which semantically indicates that all transferred underlying tokens have been fully consumed. Since cross-contract calls may fail, and function `on_burrowland_supply_client_echo()` itself may trigger an additional `MPC` signature related cross-contract call that can also fail, the minting step is not guaranteed to complete. In such cases, `Burrowland` has already finalized the conversion of underlying tokens into shares, while the user does not receive the minted shares in the `LSD` contract.

```
18  pub fn ft_on_transfer(
19      &mut self,
20      sender_id: AccountId,
21      amount: U128,
22      msg: String,
23  ) -> ContractResult<PromiseOrValue<U128>> {
24      let token_id = env::predecessor_account_id();
25      if token_id == env::current_account_id() {
26          // User initiates unstake LSD token request
27          must!(
28              !self.pa_is_paused("ft_on_transfer".to_string()),
29              GlobalError::FtOnTransferPaused
30          );
31          must!(
32              serde_json::from_str::<WormholeMsg>(&msg).is_ok()
33                  || msg.parse::<AccountId>().is_ok(),
34              GlobalError::InvalidMsg
35          );
36          let burn_lsd_amount = amount.0;
37          let burn_burrowland_shares = u128_ratio(
38              burn_lsd_amount,
39              self.underlying_burrowland_shares,
40              self.token.total_supply,
41          );
42          self.token
43              .internal_withdraw(&env::current_account_id(), burn_lsd_amount);
44          self.underlying_burrowland_shares -= burn_burrowland_shares;
45          Ok(Promise::new(self.burrowland_id.clone())
46              .function_call_weight(
47                  "client_echo_withdraw_by_shares",
```

```
                json!({
                    "token_id": self.underlying_token_id,
                    "shares": U128(burn_burrowland_shares),
                    "client_echo": msg,
                })
                .to_string()
                .into_bytes(),
                ZERO_YOCTONEAR,
                GAS_0_TGAS,
                Default::default(),
            )
            .then(
                Self::ext(env::current_account_id())
                    .with_static_gas(GAS_5_TGAS)
                    .with_unused_gas_weight(0)
                    .client_echo_withdraw_by_shares_callback(
                        sender_id,
                        U128(burn_lsd_amount),
                        U128(burn_burrowland_shares),
                    ),
            )
            .into())
    } else if token_id == self.underlying_token_id {
        if sender_id == self.burrowland_id {
            // User unstake LSD token action completed, received underlying token
            if let Ok(WormholeMsg { block_hash, msg }) =
                serde_json::from_str::<WormholeMsg>(&msg)
            {
                Ok(self
                    .internal_sign_wormhole_transfer_call_tx(
                        &token_id, amount.0, block_hash, &msg,
                    )
                    .then(
                        Self::ext(env::current_account_id())
                            .with_static_gas(GAS_5_TGAS)
                            .with_unused_gas_weight(0)
                            .sign_wormhole_transfer_call_tx_callback(amount),
                    )
                    .into())
            } else if let Ok(receiver_id) = msg.parse::<AccountId>() {
                Ok(ext_ft_core::ext_on(
                    ext_storage_management::ext(token_id)
                        .with_attached_deposit(STORAGE_DEPOSIT_YOCTONEAR)
                        .storage_deposit(Some(receiver_id.clone()), Some(true)),
                )
                .with_attached_deposit(ONE_YOCTONEAR)
                .ft_transfer(receiver_id, amount, None)
                .then(
                    Self::ext(env::current_account_id())
                        .with_static_gas(GAS_5_TGAS)
                        .with_unused_gas_weight(0)
                        .ft_transfer_callback(amount),
                )
```

```
101                 .into())
102             } else {
103                 unreachable!()
104             }
105         } else {
106             // User initiates stake underlying token request, or rebalance reward received
                    underlying token
107             must!(
108                 !self.pa_is_paused("ft_on_transfer".to_string()),
109                 GlobalError::FtOnTransferPaused
110             );
111             must!(
112                 serde_json::from_str::<WormholeMsg>(&msg).is_ok()
113                     || msg.parse::<AccountId>().is_ok(),
114                 GlobalError::InvalidMsg
115             );
116             Ok(ext_ft_core::ext(token_id)
117                 .with_attached_deposit(ONE_YOCTONEAR)
118                 .with_static_gas(GAS_30_TGAS)
119                 .ft_transfer_call(
120                     self.burrowland_id.clone(),
121                     amount,
122                     None,
123                     serde_json::to_string(&BurrowlandTokenReceiverMsg::ClientEchoDeposit {
124                         client_echo: msg,
125                     })
126                     .unwrap(),
127                 )
128                 .then(
129                     Self::ext(env::current_account_id())
130                         .with_static_gas(GAS_5_TGAS)
131                         .with_unused_gas_weight(0)
132                         .ft_transfer_call_callback(amount),
133                 )
134                 .into())
135         }
136     } else {
137         // reward token received from withdraw_reward call
138         must!(
139             sender_id == self.burrowland_id,
140             GlobalError::RewardTokenMustFromBurrowland
141         );
142         self.rewards
143             .entry(token_id)
144             .and_modify(|v| *v += amount.0)
145             .or_insert(amount.0);
146         Ok(PromiseOrValue::Value(U128(0)))
147     }
148 }
```

Listing 2.3: contracts/rhea_lsd/src/interfaces/token_receiver.rs

```
221 pub fn on_burrowland_supply_client_echo(
```

```
222         &mut self,
223         token_id: AccountId,
224         supplied_shares: U128,
225         msg: String,
226     ) -> ContractResult<()> {
227         // User stake underlying token action completed, received shares
228         must!(
229             token_id == self.underlying_token_id,
230             GlobalError::InvalidSharesTokenId
231         );
232         must!(
233             env::predecessor_account_id() == self.burrowland_id,
234             GlobalError::OnlyBurrowlandCanCall
235         );
236 
237         if let Ok(WormholeMsg { block_hash, msg }) = serde_json::from_str::<WormholeMsg>(&msg) {
238             let mint_amount = if self.token.total_supply > 0 {
239                 u128_ratio(
240                     supplied_shares.0,
241                     self.token.total_supply,
242                     self.underlying_burrowland_shares,
243                 )
244             } else {
245                 supplied_shares.0
246             };
247             self.token
248                 .internal_deposit(&env::current_account_id(), mint_amount);
249             self.underlying_burrowland_shares += supplied_shares.0;
250             FtMint {
251                 owner_id: &env::current_account_id(),
252                 amount: U128(mint_amount),
253                 memo: Some(&msg),
254             }
255             .emit();
256             self.internal_sign_wormhole_transfer_call_tx(
257                 &env::current_account_id(),
258                 mint_amount,
259                 block_hash,
260                 &msg,
261             )
262             .detach();
263         } else if let Ok(receiver_id) = msg.parse::<AccountId>() {
264             if !self.token.accounts.contains_key(&receiver_id) {
265                 self.token.internal_register_account(&receiver_id);
266             }
267             let mint_amount = if self.token.total_supply > 0 {
268                 if receiver_id == env::current_account_id() {
269                     let protocol_fee_shares =
270                         u128_ratio(supplied_shares.0, self.protocol_fee_rate as u128, BP_DENOM);
271                     let mint_protocol_fee_amount = u128_ratio(
272                         protocol_fee_shares,
273                         self.token.total_supply,
274                         self.underlying_burrowland_shares,
```

```
275                 );
276                 self.acc_protocol_fee += mint_protocol_fee_amount;
277                 mint_protocol_fee_amount
278             } else {
279                 u128_ratio(
280                     supplied_shares.0,
281                     self.token.total_supply,
282                     self.underlying_burrowland_shares,
283                 )
284             }
285         } else {
286             supplied_shares.0
287         };
288         self.token.internal_deposit(&receiver_id, mint_amount);
289         self.underlying_burrowland_shares += supplied_shares.0;
290         FtMint {
291             owner_id: &receiver_id,
292             amount: U128(mint_amount),
293             memo: if receiver_id == env::current_account_id() {
294                 Some("protocol fee")
295             } else {
296                 Some("user stake underlying token")
297             },
298         }
299         .emit();
300     } else {
301         unreachable!()
302     }
303     Ok(())
304 }
```

Listing 2.4: contracts/rhea_lsd/src/interfaces/token_receiver.rs

Impact This flaw may cause users to receive neither `LSD` tokens nor a refund when the function `on_burrowland_supply_client_echo()` fails.

Suggestion Revise the logic accordingly.

Feedback from the project `Burrowland` invokes function `on_burrowland_supply_client_echo()` with sufficient gas, so in theory the mint operation should not fail. The current fix adds the explicit failure event, enabling the backend to detect mint failures and allowing the `DAO` to uniformly recover the affected assets via function `claim_lostfound()` and subsequently transfer them to the designated account.

2.1.3 Funds locked due to lack of fallback mechanism in Wormhole `VAA` execution

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `wormhole`, the function `process_wormhole_transfer_with_payload()` finalizes a `Wormhole` bridge transfer after the `VAA` is verified and the cross-chain transfer is

reported as successful. Upon receiving this confirmation, the function immediately records the `vaa_hash` into `executed_vaas` and then invokes function `ft_transfer_call()` to transfer the bridged underlying token to the `LSD` contract itself. This transfer represents the final step of a successful cross-chain delivery and relies on the function `ft_transfer_call()` to invoke function `ft_on_transfer()` for subsequent accounting logic. However, if the execution of the function `ft_transfer_call()` or the corresponding handling of the function `ft_on_transfer()` fails or returns unused tokens, the underlying tokens remain held by the `LSD` contract while the `VAA` is already marked as executed. Since the same `VAA` cannot be processed again, the stuck tokens cannot be recovered through the `Wormhole` execution flow.

```
 95   pub fn process_wormhole_transfer_with_payload(
 96       &mut self,
 97       vaa_hash: String,
 98       token_id: AccountId,
 99       amount: U128,
100       user_message: String,
101       #[callback] res: (bool, bool),
102   ) -> ContractResult<Promise> {
103       let (is_verified, is_transferred) = res;
104       must!(is_verified && is_transferred, WormholeError::UnclaimedVaa);
105       must!(
106           self.data_mut()
107               .wormhole_state
108               .executed_vaas
109               .insert(vaa_hash.clone()),
110           WormholeError::ExecutedVaa(vaa_hash)
111       );
112       Ok(ext_ft_core::ext(token_id)
113           .with_attached_deposit(ONE_YOCTONEAR)
114           .ft_transfer_call(env::current_account_id(), amount, None, user_message))
115   }
```

Listing 2.5: contracts/rhea_lsd/src/interfaces/wormhole.rs

Impact Failures or unused returns from function `ft_on_transfer()` can strand underlying tokens in the `LSD` contract while preventing any retry for the same `VAA`.

Suggestion Delay marking the `VAA` as executed until after functions `ft_transfer_call()` and `ft_on_transfer()` successfully execute, or provide a withdrawal mechanism that allows recovery of underlying tokens when the final transfer handling fails.

2.1.4 Unrecoverable reward assets from cross-contract transfer handling

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Privileged accounts can invoke function `rebalance_reward()` to swap accrued rewards into the underlying token via `Ref-Exchange` and then deposit the corresponding underlying token into `Burrowland`. However, `Ref-Exchange` transfers the target underlying token to the `LSD` contract using a cross-contract invocation after the swap completes, which

may fail. Instead of reverting the swap and refunding the original reward asset, Ref-Exchange records the swapped assets as an internal balance claimable by the recipient contract. Although Ref-Exchange provides a withdrawal interface for contracts to recover such internally recorded assets, the LSD contract does not implement this logic. At the same time, the LSD contract relies on the function ft_transfer_call() invoked by Ref-Exchange to trigger the function ft_on_transfer() of itself, which is required to execute the deposit of underlying tokens into Burrowland. If the transfer fails, the deposit logic will never be executed, leaving the swapped rewards neither deposited nor reflected in the LSD shares value.

```
43  pub fn rebalance_reward(
44      &mut self,
45      dex_id: AccountId,
46      token_id: AccountId,
47  ) -> ContractResult<Promise> {
48      let msg = self
49          .swap_msg_template
50          .get(&swap_msg_template_key(&dex_id, &token_id))
51          .cloned()
52          .unwrap();
53      let amount = self.rewards.remove(&token_id).unwrap();
54      Ok(ext_ft_core::ext(token_id.clone())
55          .with_attached_deposit(ONE_YOCTONEAR)
56          .with_static_gas(GAS_30_TGAS)
57          .ft_transfer_call(dex_id, U128(amount), None, msg)
58          .then(
59              Self::ext(env::current_account_id())
60                  .with_static_gas(GAS_5_TGAS)
61                  .with_unused_gas_weight(0)
62                  .rebalance_reward_callback(token_id, U128(amount)),
63          ))
64  }
```

Listing 2.6: contracts/rhea_lsd/src/interfaces/rebalance.rs

```
67  pub fn rebalance_reward_callback(
68      &mut self,
69      token_id: AccountId,
70      amount: U128,
71      #[callback] used_amount: U128,
72  ) {
73      if used_amount.0 == 0 {
74          self.rewards
75              .entry(token_id)
76              .and_modify(|v| *v += amount.0)
77              .or_insert(amount.0);
78      }
79  }
```

Listing 2.7: contracts/rhea_lsd/src/interfaces/rebalance.rs

Impact Accrued rewards can become permanently unaccounted for in the LSD system, causing the reported share value to diverge from the actual value.

Suggestion Implement explicit logic in the LSD contract to recover internally recorded assets from Ref-Exchange and complete the intended deposit flow.

2.2 Recommendation

2.2.1 Add an executed VAA check in function to_submit_vaa()

Status Confirmed

Introduced by Version 1

Description In contract wormhole, the function to_submit_vaa() submits a VAA to the Wormhole bridge contract by internally invoking the function submit_vaa(). However, the function to_submit_vaa() does not verify whether the provided VAA has already been executed, which may result in redundant submissions and unnecessary gas consumption. It is recommended to check the execution status of the VAA before invoking the function submit_vaa().

```
 9   pub fn to_submit_vaa(&mut self, vaa: String) -> Promise {
10       ext_wormhole::ext(self.wormhole_state.wormhole_bridge_id.clone())
11           .with_attached_deposit(env::attached_deposit())
12           .submit_vaa(vaa, Some(env::predecessor_account_id()))
13   }
```

Listing 2.8: contracts/rhea_lsd/src/interfaces/wormhole.rs

Suggestion Add a check to verify whether the VAA has already been executed before submitting it.

2.3 Note

2.3.1 Ensure timely invocation of function rebalance() to update the share value

Introduced by Version 1

Description The LSD share value is determined by the amount of underlying tokens in Burrowland, and accrued rewards are incorporated only via the function rebalance() executed by a privileged operator. Since this function must be invoked actively and is not triggered automatically, it is important to ensure it is invoked in a timely manner to prevent new stakers from diluting existing holders and unstakers from losing their proportional rewards.

2.3.2 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles (e.g., Operator) can conduct sensitive operations, which introduces potential centralization risks. For example, a privileged account holding the contract's full access key can arbitrarily upgrade the contract code. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.3 Assumption on DEX compatibility

Introduced by Version 1

Description The LSD contract relies on the DEX to deliver converted underlying tokens via function ft_transfer_call(), which triggers function ft_on_transfer() to execute the subsequent supply of underlying tokens into Burrowland and update user share value. If a DEX transfers tokens using function ft_transfer() instead, this callback-based logic would not be executed and the rebalance flow would be incomplete. The integration therefore assumes that any supported DEX correctly uses the function ft_transfer_call() and provides the expected message payload to ensure the downstream accounting logic is triggered as intended.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS