SECURITY AUDIT REPORT

for

StaFi Cosmos LSD

Prepared By: Xiaomi Huang

PeckShield

March 10, 2024

Document Properties

Client	StaFi
Title	Security Audit Report
Target	StaFi Cosmos LSD
Version	1.0
Author	Daisy Cao
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 10, 2024	Daisy Cao	Final Release
1.0-rc	March 6, 2024	Daisy Cao	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the StaFi Cosmos LSD protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About StaFi Cosmos LSD

StaFi Cosmos LSD Stack powered by StaFi Protocol is a suite of software that helps developers deploying LSD project instantly. Thanks to Neutron, the LSD stack is are able to implement liquid staking in smart contract. The key component is StakeManager for handling staking logic, validator set management, reward distribution, and withdrawals. In addition, the LSD token is an cw20 compatible contract so that users get LST after stake and it will be burnt after unstake. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of StaFi Cosmos LSD

Item	Description
Name	StaFi
Website	https://stafi.io
Type	Cosmos
Language	Rust
Audit Method	Whitebox
Latest Audit Report	March 10, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. This audit focuses on the contracts under the stake_manager directory.

- https://github.com/stafiprotocol/neutron-lsd-contracts.git (4d9db26)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/stafiprotocol/neutron-lsd-contracts.git (2f65b2f)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `StaFi Cosmos LSD Stack` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	■■■
Low	3	■■■
Informational	0	
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Missing Status Update in execute_rm_pool_validator()	Business Logic	Resolved
PVE-002	Medium	Missing Validator Update in execute_pool_update_validator()	Business Logic	Resolved
PVE-003	Low	Improved Era Stake Logic in execute_era_stake()	Business Logic	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	Low	Improved Redelegation Logic in execute_pool_update_validator()	Business Logic	Resolved
PVE-006	Low	Redundant State/Code Removal	Coding Practices	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Missing Status Update in execute_rm_pool_validator()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: execute_rm_pool_validator.rs
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The Stake-manager contract in StaFi protocol holds all LSD functionalities. For example, there is a function named execute_rm_pool_validator() that is defined to remove a validator from the pool. In particular, if the local validator_update_status is equal to WaitQueryUpdate, there is a need to execute pool_update_validators_icq() to synchronize the contract content's ICQ with the latest validator-related queries. While reviewing the validator removal logic, we notice that validator_update_status should be properly set to WaitQueryUpdate in certain situations.

To elaborate, we show below the related code snippet from the execute_rm_pool_validator() routine. It comes to our attention that after the removal of the validator from the pool, if the validator still exists in the current delegation's query, the validator_update_status should be updated to WaitQueryUpdate for synchronization purposes (even if the delegation amount is zero).

```
pub fn execute_rm_pool_validator(
    mut deps: DepsMut<NeutronQuery>,
    info: MessageInfo,
    pool_addr: String,
    validator_addr: String,
) -> NeutronResult<Response<NeutronMsg>> {
    let mut pool_info = POOLS.load(deps.storage, pool_addr.clone())?;
    pool_info.authorize(&info.sender)?;
    pool_info.require_era_ended()?;
    pool_info.require_update_validator_ended()?;

    if !pool_info.validator_addrs.contains(&validator_addr) {
```

```
            return Err(ContractError::OldValidatorNotExist {}.into());
        }

        let delegations = query_delegation_by_addr(deps.as_ref(), pool_addr.clone())?;

        if pool_info.validator_addrs.len() <= 1 {
            return Err(ContractError::ValidatorAddressesListSize {}.into());
        }

        let left_validators: Vec<String> = pool_info
            .validator_addrs
            .clone()
            .into_iter()
            .filter(|val| val.to_string() != validator_addr)
            .collect();
        let mut rsp = Response::new();
        if let Some(to_be_redelegate_delegation) = delegations
            .delegations
            .iter()
            .find(|d| d.validator == validator_addr)
        {
            if to_be_redelegate_delegation.amount.amount.is_zero() {
                pool_info.validator_addrs = left_validators;
            } else {
                let fee = min_ntrn_ibc_fee(query_min_ibc_fee(deps.as_ref())?.min_fee);
                let (pool_ica_info, _, _) =
                    INFO_OF_ICA_ID.load(deps.storage, pool_info.ica_id.clone())?;
                    ...
            }
        }
    }
```

Listing 3.1: `execute_rm_pool_validator()`

Recommendation Change the `validator_update_status` to `WaitQueryUpdate` when the removed validator has a delegation amount of zero in above-mentioned function.

Status This issue has been fixed in the following commit: `2f65b2ff`

3.2 Missing Validator Update in execute_pool_update_validator()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: execute_pool_update_validator.rs
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the audited Stake-manager contract, we notice a function named execute_pool_update_validator() that is used to updates validator information for the pool. As mentioned in Section 3.1, another helper routine pool_update_validators_icq() is used for synchronization. And in the process, the latest validator information of the pool will be required. However, we observe that the latest validator information of the pool is not updated when validator_update_status is set to WaitQueryUpdate.

In the following, we show the related execute_pool_update_validator() function. Although appropriate actions are taken for both old_validator and new_validator, it ultimately does not get updated in the pool.

```
    pub fn execute_pool_update_validator(
    mut deps: DepsMut<NeutronQuery>,
    info: MessageInfo,
    pool_addr: String,
    old_validator: String,
    new_validator: String,
) -> NeutronResult<Response<NeutronMsg>> {
    let mut pool_info: crate::state::PoolInfo = POOLS.load(deps.storage, pool_addr.clone())?;
    pool_info.authorize(&info.sender)?;
    pool_info.require_era_ended()?;
    pool_info.require_update_validator_ended()?;

    if !pool_info.validator_addrs.contains(&old_validator) {
        return Err(ContractError::OldValidatorNotExist {}.into());
    }
    if pool_info.validator_addrs.contains(&new_validator) {
        return Err(ContractError::NewValidatorAlreadyExist {}.into());
    }

    let delegations = query_delegation_by_addr(deps.as_ref(), pool_addr.clone())?;

    let mut new_validators = pool_info.validator_addrs.clone();
    new_validators.retain(|x| x.as_str() != old_validator);
```

```
    new_validators.push(new_validator.clone());

    let mut msgs = vec![];

    for delegation in delegations.delegations {
        if delegation.validator != old_validator {
            continue;
        }
        let stake_amount = delegation.amount.amount;

        if stake_amount.is_zero() {
            break;
        }

        let any_msg = gen_redelegate_txs(
            pool_addr.clone(),
            delegation.validator.clone(),
            new_validator.clone(),
            pool_info.remote_denom.clone(),
            stake_amount,
        );

        msgs.push(any_msg);
    }
    let (pool_ica_info, _, _) = INFO_OF_ICA_ID.load(deps.storage, pool_info.ica_id.clone
        ())?;

    // let remove_msg_old_query = NeutronMsg::remove_interchain_query(registere_query_id
        );
    let mut resp = Response::default(); // .add_message(remove_msg_old_query)

    if !msgs.is_empty() {
        ...
    } else {
        pool_info.validator_update_status = ValidatorUpdateStatus::WaitQueryUpdate;
    }

    POOLS.save(deps.storage, pool_addr.clone(), &pool_info)?;

    Ok(resp)
}
```

Listing 3.2: `execute_pool_update_validator()`

Recommendation Revise the above routine to properly update latest validator information of the pool.

Status This issue has been fixed in the following commit: `2f65b2ff`

3.3 Improved Era Stake Logic in execute_era_stake()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `execute_era_stake.rs`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Stake-manager` contract introduces the concept of `era`. The new `era` process is permissionless and showcases the decentralized nature of the `Cosmos LSD Stack` by allowing anyone to trigger the beginning of a new `era`. Each step in the process includes necessary validations to prevent the contract from re-processing transactions or prematurely moving to subsequent steps. `era_stake` is one of the steps to handle staking, unstaking, and withdrawal transactions on the original chain.

While reviewing its logic, we notice that when the bonded amount is greater than the unbounded amount, it will be evenly delegated to each validator. However, if the total amount is small, it will be more gas-efficient by delegating to a single validator.

```
    pub fn execute_era_stake(
        mut deps: DepsMut<NeutronQuery>,
        env: Env,
        pool_addr: String,
    ) -> NeutronResult<Response<NeutronMsg>> {
        let mut pool_info = POOLS.load(deps.storage, pool_addr.clone())?;

        // check era state
        if pool_info.status != EraUpdateEnded {
            return Err(ContractError::StatusNotAllow {}.into());
        }

        let mut msgs = vec![];

        let mut msg_str = "".to_string();
        if pool_info.era_snapshot.unbond >= pool_info.era_snapshot.bond {
            ...} else {
            let stake_amount = pool_info.era_snapshot.bond - pool_info.era_snapshot.unbond;
            let validator_count = pool_info.validator_addrs.len() as u128;
            if validator_count == 0 {
                return Err(ContractError::ValidatorsEmpty {}.into());
            }

            let amount_per_validator = stake_amount.div(Uint128::from(validator_count));
            let remainder = stake_amount.sub(amount_per_validator.mul(Uint128::new(validator_count)));

```

```
            for (index, validator_addr) in pool_info.validator_addrs.iter().enumerate() {
                let mut amount_for_this_validator = amount_per_validator;

                // Add the remainder to the first validator
                if index == 0 {
                    amount_for_this_validator += remainder;
                }

                let any_msg = gen_delegation_txs(
                    pool_addr.clone(),
                    validator_addr.clone(),
                    pool_info.remote_denom.clone(),
                    amount_for_this_validator,
                );

                msgs.push(any_msg);
            }
        }

        if msgs.len() == 0 {
            pool_info.status = EraStakeEnded;
            POOLS.save(deps.storage, pool_addr, &pool_info)?;

            return Ok(Response::default());
        }

        let (pool_ica_info, _, _) = INFO_OF_ICA_ID.load(deps.storage, pool_info.ica_id.clone())?;

        let fee = min_ntrn_ibc_fee(query_min_ibc_fee(deps.as_ref())?.min_fee);
        let cosmos_msg = NeutronMsg::submit_tx(
            pool_ica_info.ctrl_connection_id,
            pool_info.ica_id.clone(),
            msgs,
            "".to_string(),
            DEFAULT_TIMEOUT_SECONDS,
            fee,
        );

        let submsg = msg_with_sudo_callback(
            deps.branch(),
            cosmos_msg,
            SudoPayload {
                port_id: pool_ica_info.ctrl_port_id,
                // the acknowledgement later
                message: msg_str,
                pool_addr: pool_addr.clone(),
                tx_type: TxType::EraBond,
            },
        )?;

```

```
        pool_info.status = EraStakeStarted;
        POOLS.save(deps.storage, pool_addr, &pool_info)?;

        Ok(Response::default().add_submessage(submsg))
    }
```

Listing 3.3: `execute_era_stake()`

Recommendation Delegate to a single validator when `stake_amount` is below a threshold in the above function.

Status This issue has been fixed in the following commit: `2f65b2ff`

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the audited protocol, there is a privileged account, i.e., `admin`. This account plays a critical role in regulating the protocol-wide operations (e.g., configure stack, pool parameters). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the market contract as an example and show the representative functions potentially affected by the privileges of the `admin` account.

```
    pub fn execute_config_stack(
    deps: DepsMut<NeutronQuery>,
    info: MessageInfo,
    param: ConfigStackParams,
) -> NeutronResult<Response<NeutronMsg>> {
    let mut stack = STACK.load(deps.storage)?;
    stack.authorize(&info.sender)?;

    if let Some(stack_fee_receiver) = param.stack_fee_receiver {
        stack.stack_fee_receiver = stack_fee_receiver
    }
    if let Some(stack_fee_commission) = param.stack_fee_commission {
        stack.stack_fee_commission = stack_fee_commission;
    }
    if let Some(new_admin) = param.new_admin {
        stack.admin = new_admin;
    }
```

```
    if let Some(lsd_token_code_id) = param.lsd_token_code_id {
        stack.lsd_token_code_id = lsd_token_code_id;
    }
    if let Some(add_entrusted_pool) = param.add_entrusted_pool {
        if !stack.entrusted_pools.contains(&add_entrusted_pool) {
            stack.entrusted_pools.push(add_entrusted_pool);
        }
    }
    if let Some(remove_entrusted_pool) = param.remove_entrusted_pool {
        if stack.entrusted_pools.contains(&remove_entrusted_pool) {
            stack
                .entrusted_pools
                .retain(|p| p.to_string() != remove_entrusted_pool);
        }
    }

    STACK.save(deps.storage, &stack)?;

    Ok(Response::default())
}
```

Listing 3.4: `execute_config_stack()`

We understand the need of the privileged functions for proper operations, but at the same time the extra power to the `admin` may also be a counter-party risk to the `StaFi` users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to `StaFi` explicit to `StaFi` users.

Status The issue has been confirmed by the team.

3.5 Improved Redelegation Logic in execute_pool_update_validator()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: execute_pool_update_validator.rs
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [3]

Description

The Stake-manager contract provides a function named execute_pool_update_validator() that is used to updates validator information for the pool. While examining the routine, we notice the current implementation can be improved.

To elaborate, we show below the code snippet from this execute_pool_update_validator() function. It comes to our attention that if the following condition is met, i.e., stake_amount.is_zero() (line 618), the for-loop execution should break instead of current continue (line 619).

```
600     pub fn execute_pool_update_validator(
601     mut deps: DepsMut<NeutronQuery>,
602     info: MessageInfo,
603     pool_addr: String,
604     old_validator: String,
605     new_validator: String,
606 ) -> NeutronResult<Response<NeutronMsg>> {
607     let mut pool_info: crate::state::PoolInfo = POOLS.load(deps.storage, pool_addr.clone())?;
608     pool_info.authorize(&info.sender)?;
609     pool_info.require_era_ended()?;
610     pool_info.require_update_validator_ended()?;
611     ...
612     for delegation in delegations.delegations {
613         if delegation.validator != old_validator {
614             continue;
615         }
616         let stake_amount = delegation.amount.amount;
617
618         if stake_amount.is_zero() {
619             continue;
620         }
621
622         let any_msg = gen_redelegate_txs(
623             pool_addr.clone(),
624             delegation.validator.clone(),
625             new_validator.clone(),
626             pool_info.remote_denom.clone(),
627             stake_amount,
```

```
        );

        msgs.push(any_msg);
    }
    let (pool_ica_info, _, _) = INFO_OF_ICA_ID.load(deps.storage, pool_info.ica_id.clone
        ())?;

    // let remove_msg_old_query = NeutronMsg::remove_interchain_query(registere_query_id
        );
    let mut resp = Response::default(); // .add_message(remove_msg_old_query)
    ...
    POOLS.save(deps.storage, pool_addr.clone(), &pool_info)?;

    Ok(resp)
}
```

Listing 3.5: `execute_pool_update_validator()`

Recommendation Exit the `for`-loop execution if the condition of `stake_amount.is_zero()` is satisfied.

Status The issue has been fixed by this commit: `2f65b2ff`

3.6 Redundant State/Code Removal

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `error_conversion.rs`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

In the `Stake-manager` contract, the `error_conversion` file defines all the error codes used in the contract. Error codes enables developers and users to better understand and address runtime exceptions. While examining its logic, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

To elaborate, we show below the related code snippet of the `error_conversion.rc` file. There are two error codes, `RateIsZero` and `RateNotMatch`, that are not used throughout the entire contract and can be safely removed.

```
    pub enum ContractError {

        #[error("Encode error: {0}")]
        EncodeError(String),
```

```
        #[error("Validator for unbond not enough")]
        ValidatorForUnbondNotEnough {},

        #[error("Delegation submission height")]
        DelegationSubmissionHeight {},

        #[error("Withdraw Addr balances submission height")]
        WithdrawAddrBalanceSubmissionHeight {},

        #[error("Rebond height")]
        RebondHeight {},

        #[error("Pool is paused")]
        PoolIsPaused {},

        #[error("Already latest era")]
        AlreadyLatestEra {},

        #[error("Validator addresses list")]
        ValidatorAddressesListSize {},

        #[error("Rate is zero")]
        RateIsZero {},

        #[error("Instantiate2 address failed, err: {0}")]
        Instantiate2AddressFailed(String),

        #[error("Rate not match")]
        RateNotMatch {},

        #[error("Closed channel ID unmatch")]
        ClosedChannelIdUnmatch {},
        ...
    }
```

Listing 3.6: `error_conversion.rs`

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status The issue has been fixed by this commit: `2f65b2ff`

4 Conclusion

In this audit, we have analyzed the design and implementation of the `StaFi Cosmos LSD Stack` that is a suite of software to helps developers deploying `LSD` project instantly. Thanks to `Neutron`, the `LSD stack` is are able to implement liquid staking in smart contract. The key component is `StakeManager` for handling staking logic, validator set management, reward distribution, and withdrawals. In addition, the `LSD` token is an `cw20` compatible contract so that users get `LST` after stake and it will be burnt after unstake. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.