SECURITY AUDIT REPORT

for

Stafi rDEX

Prepared By: Yiqun Chen

PeckShield

January 21, 2022

Document Properties

Client	Stafi Protocol
Title	Security Audit Report
Target	Stafi rDEX
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 21, 2022	Xiaotao Wu	Final Release
1.0-rc	January 10, 2022	Xiaotao Wu	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the Stafi rDEX 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 rDEX

rDEX is a DEX platform running on the Stafi chain, which aims to enable users to directly trade rToken and FIS with each other. Liquidity providers can obtain pool tokens when liquidity is provided and also earn the fees incurred in the process of user trades. Users can also participate in LP mining to obtain rewards. Moreover, the protocol provides the LP compensation logic to compensate users for possible losses during mining. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Stafi rDEX

Item	Description
Name	Stafi Protocol
Website	https://stafi.io/
Type	Stafi Blockchain
Platform	Rust
Audit Method	Whitebox
Latest Audit Report	January 21, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that only the rdex module in the scope of this audit.

- https://github.com/stafiprotocol/stafi-node/tree/rswap (0f924f2)

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

- https://github.com/stafiprotocol/stafi-node/tree/rswap (3d7e08a)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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.

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

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

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) [9], 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`'s `rDEX` 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	1	■
Medium	1	■
Low	1	■
Informational	1	■
Total	4	

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 1 high-severity vulnerability, 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible User Asset Loss In swap::add_liquidity()	Time and State	Confirmed
PVE-002	High	Possible Swap Loss In swap::remove_liquidity()	Time and State	Fixed
PVE-003	Informational	Improved Logic In swap::add_liquidity()	Business Logic	Fixed
PVE-004	Medium	Trust Issue of Admin Keys)	Security Features	Confirmed

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 Possible User Asset Loss In swap::add_liquidity()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `node/pallets/rdex/swap/src/lib.rs`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [2]

Description

The `rDEX swap` module allows users to add liquidity to an existing `rToken/FIS` pool and get in return the corresponding `LP` tokens. While examining the current implementation, we notice an issue that may cause user asset loss.

To elaborate, we show below the `add_liquidity()` routine. This routine is used for participating users to add the supported `rToken/FIS` as liquidity to an existing pool and get `LP` tokens in return. The issue occurs when the `total_unit` of the pool is equal to 0 under the assumption that all liquidity providers have removed liquidity from the pool.

```
/// add liquidity
#[weight = 10_000_000_000]
pub fn add_liquidity(origin, symbol: RSymbol, rtoken_amount: u128, fis_amount: u128)
    -> DispatchResult {
    let who = ensure_signed(origin)?;
    let mut pool = Self::swap_pools(symbol).ok_or(Error::<T>::PoolNotExist)?;

    ensure!(fis_amount > 0 || rtoken_amount > 0, Error::<T>::AmountAllZero);
    ensure!(T::RCurrency::free_balance(&who, symbol) >= rtoken_amount, Error::<T>::
        UserRTokenAmountNotEnough);
    ensure!(T::Currency::free_balance(&who).saturated_into::<u128>() > fis_amount,
        Error::<T>::UserFisAmountNotEnough);

    let (new_total_pool_unit, add_lp_unit) = Self::cal_pool_unit(pool.total_unit,
        pool.fis_balance, pool.rtoken_balance, fis_amount, rtoken_amount);
```

```
135
136         // transfer token to module account
137         T::Currency::transfer(&who, &Self::account_id(), fis_amount.saturated_into(),
                KeepAlive)?;
138         T::RCurrency::transfer(&who, &Self::account_id(), symbol, rtoken_amount)?;
139
140         // update pool
141         pool.total_unit = new_total_pool_unit;
142         pool.fis_balance =  pool.fis_balance.saturating_add(fis_amount);
143         pool.rtoken_balance = pool.rtoken_balance.saturating_add(rtoken_amount);
144
145         // update pool/lp storage
146         T::LpCurrency::mint(&who, symbol, add_lp_unit)?;
147         <SwapPools>::insert(symbol, pool.clone());
148         Self::deposit_event(RawEvent::AddLiquidity(who, symbol, fis_amount,
                rtoken_amount, new_total_pool_unit, add_lp_unit, pool.fis_balance, pool.
                rtoken_balance));
149         Ok(())
150     }
```

Listing 3.1: `node/pallets/rdex/swap/src/lib.rs`

Specifically, if all liquidity providers have removed liquidity from the pool, the `total_unit` of the pool is equal to 0. Then if a user calls the `add_liquidity()` function to add single asset to the pool, the calculated `add_lp_unit` will be 0 (lines 257-262) and the mint amount for this user will also be 0 (line 146) since `add_lp_unit == 0`. Thus the user will suffer asset losses when calling the `add_liquidity()` function. Another issue is that when a user add two assets to an empty pool, the minted `LP` amount only depends on the amount of `FIS` this user added to the pool, regardless of the amount of `rToken` added by the user (lines 263-265).

```
240     // F = fis Balance (before)
241     // R = rToken Balance (before)
242     // f = fis added;
243     // r = rToken added
244     // P = existing Pool Units
245     // slipAdjustment = (1 - ABS((F r - f R)/((f + F) (r + R))))
246     // units = ((P (r F + R f))/(2 R F))*slipAdjustment
247     pub fn cal_pool_unit(
248         old_pool_unit: u128,
249         fis_balance: u128,
250         rtoken_balance: u128,
251         fis_amount: u128,
252         rtoken_amount: u128,
253     ) -> (u128, u128) {
254         if fis_amount == 0 && rtoken_amount == 0 {
255             return (0, 0);
256         }
257         if fis_balance.saturating_add(fis_amount) == 0 {
258             return (0, 0);
259         }
260         if rtoken_balance.saturating_add(rtoken_amount) == 0 {
```

```
            return (0, 0);
        }
        if fis_balance == 0 || rtoken_balance == 0 {
            return (fis_amount, fis_amount);
        }

        let p_capital = U512::from(old_pool_unit);
        let f_capital = U512::from(fis_balance);
        let r_capital = U512::from(rtoken_balance);
        let f = U512::from(fis_amount);
        let r = U512::from(rtoken_amount);

        let numerator = f_capital
            .saturating_mul(r)
            .saturating_add(f.saturating_mul(r_capital));
        let raw_unit = p_capital
            .saturating_mul(numerator)
            .checked_div(
                r_capital
                    .saturating_mul(f_capital)
                    .saturating_mul(U512::from(2)),
            )
            .unwrap_or(U512::zero());
        if raw_unit.is_zero() {
            return (0, 0);
        }

        let abs: U512;
        if f_capital.saturating_mul(r) > f.saturating_mul(r_capital) {
            abs = f_capital
                .saturating_mul(r)
                .saturating_sub(f.saturating_mul(r_capital));
        } else {
            abs = f
                .saturating_mul(r_capital)
                .saturating_sub(f_capital.saturating_mul(r));
        }

        let mut adj_unit = U512::zero();
        if !abs.is_zero() {
            let slip_adj_denominator = f
                .saturating_add(f_capital)
                .saturating_mul(r.saturating_add(r_capital));

            adj_unit = raw_unit
                .saturating_mul(abs)
                .checked_div(slip_adj_denominator)
                .unwrap_or(U512::zero());
        }

        let add_unit = raw_unit.saturating_sub(adj_unit);
        let total_unit = p_capital.saturating_add(add_unit);
```

```
        (Self::safe_to_u128(total_unit), Self::safe_to_u128(add_unit))
    }
```

Listing 3.2: `node/pallets/rdex/swap/src/lib.rs`

Recommendation Revise the above `add_liquidity()` routine to defensively calculate the mint amount when the `total_unit` of the pool is equal to 0.

Status This issue has been confirmed. The `Stafi rDEX` team has modified the code which requires the liquidity provider must add two assets when the pool is empty. As for the issue that the minted `LP` amount only depends on the amount of `FIS` a user provided when the pool is empty, the team confirms that `rDEX` encourages users to add two assets as liquidity which have equal value.

3.2 Possible Swap Loss In swap::remove_liquidity()

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `node/pallets/rdex/swap/src/lib.rs`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

The `rDEX swap` module also provides a `remove_liquidity()` function for users to remove liquidity from an existing `rToken/FIS` pool. While removing liquidity, users can also swap one asset for another via the same pool.

```
    /// remove liquidity
    #[weight = 10_000_000_000]
    pub fn remove_liquidity(origin, symbol: RSymbol, rm_unit: u128, swap_unit: u128,
        input_is_fis: bool) -> DispatchResult {
        let who = ensure_signed(origin)?;
        let mut pool = Self::swap_pools(symbol).ok_or(Error::<T>::PoolNotExist)?;
        let lp_unit = T::LpCurrency::free_balance(&who, symbol);
        let pool_fis_balance = T::Currency::free_balance(&Self::account_id()).
            saturated_into::<u128>();
        let pool_rtoken_balance = T::RCurrency::free_balance(&Self::account_id(), symbol
            );

        ensure!(rm_unit > 0 && rm_unit <= lp_unit && rm_unit >= swap_unit, Error::<T>::
            UnitAmountImproper);
```

```
        let (mut rm_fis_amount, mut rm_rtoken_amount, swap_input_amount) = Self::
            cal_remove_result(pool.total_unit, rm_unit, swap_unit, pool.fis_balance,
            pool.rtoken_balance, input_is_fis);
        //update pool/lp
        pool.total_unit = pool.total_unit.saturating_sub(rm_unit);
        pool.fis_balance =  pool.fis_balance.saturating_sub(rm_fis_amount);
        pool.rtoken_balance = pool.rtoken_balance.saturating_sub(rm_rtoken_amount);
        if swap_input_amount > 0 {
            let (swap_result, _) = Self::cal_swap_result(pool.fis_balance, pool.
                rtoken_balance, swap_input_amount, input_is_fis);
            if input_is_fis {
                pool.fis_balance = pool.fis_balance.saturating_add(swap_input_amount);
                pool.rtoken_balance = pool.rtoken_balance.saturating_sub(swap_result);

                rm_fis_amount = rm_fis_amount.saturating_sub(swap_input_amount);
                rm_rtoken_amount = rm_rtoken_amount.saturating_add(swap_result);
            } else {
                pool.rtoken_balance = pool.rtoken_balance.saturating_add(
                    swap_input_amount);
                pool.fis_balance = pool.fis_balance.saturating_sub(swap_result);

                rm_rtoken_amount = rm_rtoken_amount.saturating_sub(swap_input_amount);
                rm_fis_amount = rm_fis_amount.saturating_add(swap_result);
            }
        }

        ensure!(pool_fis_balance >= rm_fis_amount, Error::<T>::PoolFisBalanceNotEnough);
        ensure!(pool_rtoken_balance >= rm_rtoken_amount, Error::<T>::
            PoolRTokenBalanceNotEnough);

        // transfer token to user
        if rm_fis_amount > 0 {
            T::Currency::transfer(&Self::account_id(), &who, rm_fis_amount.
                saturated_into(), KeepAlive)?;
        }
        if rm_rtoken_amount > 0 {
            T::RCurrency::transfer(&Self::account_id(), &who, symbol, rm_rtoken_amount)
                ?;
        }
        // burn unit
        T::LpCurrency::burn(&who, symbol, rm_unit)?;
        // update pool
        <SwapPools>::insert(symbol, pool.clone());
        Self::deposit_event(RawEvent::RemoveLiquidity(who, symbol, rm_unit, swap_unit,
            rm_fis_amount, rm_rtoken_amount, input_is_fis, pool.fis_balance, pool.
            rtoken_balance));
        Ok(())
    }
```

Listing 3.3: `node/pallets/rdex/swap/src/lib.rs`

To elaborate, we show above the `remove_liquidity()` routine. We notice the current implementation does not specify any restriction on possible slippage for the actual swap result `swap_result` and may cause loss of user assets (line 169).

Recommendation Add necessary slippage control for above mentioned issue.

Status This issue has been fixed in the following commit: `156cf30`.

3.3 Improved Logic In swap::add_liquidity()

- ID: PVE-003
- Severity: Informational
- Likelihood: High
- Impact: Low
- Target: `node/pallets/rdex/swap/src/lib.rs`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, the `rDEX swap` module allows users to add liquidity to an existing `rToken/FIS` pool and get in return the corresponding `LP` tokens. While examining the `add_liquidity()` routine, we notice the current implementation logic can be improved.

To elaborate, we show below its code snippet. It comes to our attention that both `rToken` and `FIS` `transfer` will be called even if a user only adds a single asset to an existing pool (lines 137-138).

```
124     /// add liquidity
125     #[weight = 10_000_000_000]
126     pub fn add_liquidity(origin, symbol: RSymbol, rtoken_amount: u128, fis_amount: u128)
            -> DispatchResult {
127         let who = ensure_signed(origin)?;
128         let mut pool = Self::swap_pools(symbol).ok_or(Error::<T>::PoolNotExist)?;
129
130         ensure!(fis_amount > 0 || rtoken_amount > 0, Error::<T>::AmountAllZero);
131         ensure!(T::RCurrency::free_balance(&who, symbol) >= rtoken_amount, Error::<T>::
                UserRTokenAmountNotEnough);
132         ensure!(T::Currency::free_balance(&who).saturated_into::<u128>() > fis_amount,
                Error::<T>::UserFisAmountNotEnough);
133
134         let (new_total_pool_unit, add_lp_unit) = Self::cal_pool_unit(pool.total_unit,
                pool.fis_balance, pool.rtoken_balance, fis_amount, rtoken_amount);
135
136         // transfer token to module account
137         T::Currency::transfer(&who, &Self::account_id(), fis_amount.saturated_into(),
                KeepAlive)?;
138         T::RCurrency::transfer(&who, &Self::account_id(), symbol, rtoken_amount)?;
139
140         // update pool
```

```
        pool.total_unit = new_total_pool_unit;
        pool.fis_balance =  pool.fis_balance.saturating_add(fis_amount);
        pool.rtoken_balance = pool.rtoken_balance.saturating_add(rtoken_amount);

        // update pool/lp storage
        T::LpCurrency::mint(&who, symbol, add_lp_unit)?;
        <SwapPools>::insert(symbol, pool.clone());
        Self::deposit_event(RawEvent::AddLiquidity(who, symbol, fis_amount,
            rtoken_amount, new_total_pool_unit, add_lp_unit, pool.fis_balance, pool.
            rtoken_balance));
        Ok(())
    }
```

Listing 3.4: `node/pallets/rdex/swap/src/lib.rs`

Recommendation Only call the related `transfer()` routine if the added amount is greater than 0.

Status This issue has been fixed in the following commit: `156cf30`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `node/pallets/rdex/mining/src/lib.rs`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the `Stafi rDEX` protocol, there is certain privileged account, i.e., `root`. When examining the related files, we notice inherent trust on this privileged account. To elaborate, we show below the related functions.

Firstly, the privileged functions of the `mining` module allow for the `root` to add/remove pools and increase the pool index.

```
    /// create pool
    #[weight = 10_000]
    pub fn add_pool(origin, symbol: RSymbol, pool_index: u32, start_block: u32,
        lp_locked_blocks: u32, reward_per_block: u128, total_reward: u128,
        guard_impermanent_loss: bool) -> DispatchResult {
        ensure_root(origin.clone())?;
        let mut stake_pool_vec = Self::stake_pools((symbol, pool_index)).ok_or(Error::<T
            >::StakePoolNotExist)?;
```

```
        let current_block_num = system::Module::<T>::block_number().saturated_into::<u32
            >();
        let last_reward_block = if current_block_num < start_block {
            start_block
        } else {
            current_block_num
        };

        let stake_pool = StakePool {
            symbol: symbol,
            emergency_switch: false,
            total_stake_lp: 0,
            start_block: start_block,
            reward_per_block: reward_per_block,
            total_reward: total_reward,
            left_reward: total_reward,
            lp_locked_blocks: lp_locked_blocks,
            last_reward_block: last_reward_block,
            reward_per_share: 0,
            guard_impermanent_loss: guard_impermanent_loss,
        };
        stake_pool_vec.push(stake_pool);
        let grade_index = stake_pool_vec.len() as u32 - 1;
        <StakePools>::insert((symbol, pool_index), stake_pool_vec);
        Self::deposit_event(RawEvent::AddPool(symbol, pool_index, grade_index,
            start_block, lp_locked_blocks, reward_per_block, total_reward,
            guard_impermanent_loss));
        Ok(())
    }

    /// remove pool
    #[weight = 10_000]
    pub fn rm_pool(origin, symbol: RSymbol, pool_index: u32, grade_index: u32) ->
        DispatchResult {
        ensure_root(origin.clone())?;
        let mut stake_pool_vec = Self::stake_pools((symbol, pool_index)).ok_or(Error::<T
            >::StakePoolNotExist)?;
        let stake_pool = *stake_pool_vec.get(grade_index as usize).ok_or(Error::<T>::
            GradeIndexOverflow)?;
        ensure!(stake_pool.total_stake_lp == 0, Error::<T>::LpBalanceNotEmpty);

        stake_pool_vec.remove(grade_index as usize);
        <StakePools>::insert((symbol, pool_index), stake_pool_vec);
        Self::deposit_event(RawEvent::RmPool(symbol, pool_index, grade_index));
        Ok(())
    }

    /// increase pool index
    #[weight = 10_000]
    pub fn increase_pool_index(origin, symbol: RSymbol) -> DispatchResult {
        ensure_root(origin.clone())?;
        let pool_count = Self::pool_count(symbol);
```

```

        <StakePools>::insert((symbol, pool_count), Vec::<StakePool>::new());
        <PoolCount>::insert(symbol, pool_count + 1);

        Ok(())
    }
```

Listing 3.5: node/pallets/rdex/mining/src/lib.rs

Secondly, the privileged function of the mining module allows for the root to emergency switch an existing staking pool. The stakers can only call the deposit()/withdraw()/claim_reward() functions when the emergency_switch is set to false.

```
    /// emergency switch
    #[weight = 10_000]
    pub fn emergency_switch(origin, symbol: RSymbol, pool_index: u32, grade_index: u32) -> DispatchResult {
        ensure_root(origin.clone())?;

        let mut stake_pool_vec = Self::stake_pools((symbol, pool_index)).ok_or(Error::<T>::StakePoolNotExist)?;
        let mut stake_pool = *stake_pool_vec.get(grade_index as usize).ok_or(Error::<T>::GradeIndexOverflow)?;

        stake_pool.emergency_switch = !stake_pool.emergency_switch;
        stake_pool_vec[grade_index as usize] = stake_pool;

        <StakePools>::insert((symbol, pool_index), stake_pool_vec);

        Ok(())
    }
```

Listing 3.6: node/pallets/rdex/mining/src/lib.rs

Lastly, the privileged functions of the mining module allow for the root to withdraw the guard fund, set the guard line, and set the guard reserve.

```
    /// withdraw guard fund
    #[weight = 100_000]
    fn withdraw_guard_fund(origin, symbol: RSymbol, to_address: T::AccountId, amount: u128) -> DispatchResult {
        ensure_root(origin)?;
        let mut withdraw_amount = amount;
        let guard_reserve = Self::guard_reserve(symbol);
        if withdraw_amount > guard_reserve {
            withdraw_amount = guard_reserve;
        }
        let module_free_balance = T::Currency::free_balance(&Self::account_id()).saturated_into::<u128>();
        if withdraw_amount > module_free_balance {
            withdraw_amount = module_free_balance;
        }
        if withdraw_amount > 0 {
```

```
            T::Currency::transfer(&Self::account_id(), &to_address, withdraw_amount.
                saturated_into(), KeepAlive)?;
        }
        <GuardReserve>::insert(symbol, guard_reserve.saturating_sub(withdraw_amount));
        Ok(())
    }

    /// set guard line
    #[weight = 100_000]
    fn set_guard_line(origin, symbol: RSymbol, pool_index: u32, line: u32) ->
        DispatchResult {
        ensure_root(origin)?;
        <GuardLine>::insert((symbol, pool_index), line);
        Ok(())
    }
    /// set guard reserve
    #[weight = 100_000]
    fn set_guard_reserve(origin, symbol: RSymbol, amount: u128) -> DispatchResult {
        ensure_root(origin)?;
        <GuardReserve>::insert(symbol, amount);
        Ok(())
    }
}
```

Listing 3.7: `node/pallets/rdex/mining/src/lib.rs`

We understand the need of the privileged functions for proper `rDEX` operations, but at the same time the extra power to the `root` may also be a counter-party risk to the `rDEX` 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 `root` explicit to `Stafi` `rDEX` users.

Status This issue has been confirmed. The `Stafi` team confirms that this `root` account mainly controls some system parameters and switches.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Stafi rDEX` protocol. `rDEX` is a `DEX` platform running on the `Stafi` chain, which aims to enable users to directly trade `rToken` and `FIS` with each other. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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