BLOCKSEC

Security Audit
Report for Richswap

Date: Mar 11, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Omnity Network
Target	Richswap

Version History

Version	Date	Description
1.0	Mar 11, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-03-13 02:47:08+0000

Chapter 1 Introduction

1.1 About Target Contracts

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

The focus of this audit is the `src/lib.rs`, `src/canister.rs`, `src/pool.rs`, and `src/psbt.rs` files within the Richswap of Omnity Network [1]. Please note that these files are the only ones within the scope of our audit. While all other files in the repository are considered reliable in terms of both functionality and security, these files are not included in the scope of the audit.

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

Project	Version	Commit Hash
Richswap	`Version 1`	`ca737b6f53ff83014ca5d2f3e770fd625dc3e06c`
	`Version 2`	`582f6877b8ba4c8d47bde1d252011b973a7848e6`

1.2 Disclaimer

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

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

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

[1] https://github.com/octopus-network/richswap-canister

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

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

We show the main concrete checkpoints in the following.

1.3.1 Software Security

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

1.3.2 DeFi Security

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

1.3.3 NFT Security

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

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

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

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [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 four categories:

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

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

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

Chapter 2 Findings

In total, we find **seven** potential issues. Besides, we also have **three** recommendations and **three** notes.

- High Risk: 2
- Medium Risk: 5
- Recommendation: 3
- Note: 3

ID	Severity	Description	Category	Status
1	High	Incorrect logic of adding liquidity to an empty pool	DeFi Security	Fixed
2	High	PSBT signing fails due to missing POOL_ADDR registration	DeFi Security	Fixed
3	Medium	Incorrect rounding direction in swap and withdraw calculations	DeFi Security	Confirmed
4	Medium	Underestimation of user's share amount due to precision loss	DeFi Security	Fixed
5	Medium	Potential cycle drain due to lack of access control in function `create()`	DeFi Security	Confirmed
6	Medium	Failure of extracting protocol fees in function `available_to_extract()`	DeFi Security	Fixed
7	Medium	Incorrect check in function `available_to_swap()`	DeFi Security	Fixed
8	-	Redundant code	Recommendation	Confirmed
9	-	Add crate visibility modifier to function `validate_extract_fee()`	Recommendation	Confirmed
10	-	Comments typo correction	Recommendation	Fixed
11	-	Potential centralization risk	Note	-
12	-	Validated parameter of orchestrator	Note	-
13	-	Potential DoS in function `rollback()`	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Incorrect logic of adding liquidity to an empty pool

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There is a logical flaw in how the protocol processes liquidity additions to an empty pool. Specifically, the function `validate_adding_liquidity()` verifies user input and returns a `PoolState` structure, which is then recorded via function `commit()`. However, when handling an empty pool, the function `liquidity_should_add()` incorrectly returns 0 for the amount of tokens should be added, contradicting the intended logic.

Additionally, when calculating the updated pool balance after liquidity is added, the function fails to consider the empty state, leading to an incorrect `UTXO` balance update. Since state.utxo is `None` at this stage, the implementation fails to calculate and store the new state in `state.utxo`, resulting in an unintended deviation from the expected pool behavior.

```
226   pub(crate) fn validate_adding_liquidity(
227       &self,
228       txid: Txid,
229       nonce: u64,
230       pool_utxo_spend: Vec<String>,
231       pool_utxo_receive: Vec<String>,
232       input_coins: Vec<InputCoin>,
233       output_coins: Vec<OutputCoin>,
234       initiator: String,
235   ) -> Result<(PoolState, Option<Utxo>), ExchangeError> {
236       (input_coins.len() == 2 && output_coins.is_empty())
237           .then(|| ())
238           .ok_or(ExchangeError::InvalidSignPsbtArgs(
239               "invalid input/output_coins, add_liquidity requires 2 inputs and 0 output"
240                   .to_string(),
241           ))?;
242       let x = input_coins[0].coin.clone();
243       let y = input_coins[1].coin.clone();
244       let mut state = self.states.last().cloned().unwrap_or_default();
245       // check nonce matches
246       (state.nonce == nonce)
247           .then(|| ())
248           .ok_or(ExchangeError::PoolStateExpired(state.nonce))?;
249       // check prev_outpoint matches
250       let pool_utxo = state.utxo.clone();
251       (pool_utxo.as_ref().map(|u| u.outpoint()).as_ref() == pool_utxo_spend.last())
252           .then(|| ())
253           .ok_or(ExchangeError::InvalidSignPsbtArgs(
254               "pool_utxo_spend/pool state mismatch".to_string(),
255           ))?;
256       // check output exists
257       let pool_new_outpoint = pool_utxo_receive.last().map(|s| s.clone()).ok_or(
258           ExchangeError::InvalidSignPsbtArgs("pool_utxo_receive not found".to_string()),
259       )?;
260       // check input coins
261       let (btc_input, rune_input) = if x.id == CoinId::btc() && y.id != CoinId::btc() {
262           Ok((x, y))
263       } else if x.id != CoinId::btc() && y.id == CoinId::btc() {
264           Ok((y, x))
265       } else {
266           Err(ExchangeError::InvalidSignPsbtArgs(
267               "Invalid inputs: requires 2 different input coins".to_string(),
268           ))
269       }?;
270       // check minial liquidity
271       (btc_input.value >= MIN_BTC_VALUE as u128)
272           .then(|| ())
273           .ok_or(ExchangeError::TooSmallFunds)?;
```

```
274         // y = f(x), x' = f(y'); => x == x' || y == y'
275         let rune_expecting = self.liquidity_should_add(btc_input)?;
276         let btc_expecting = self.liquidity_should_add(rune_input)?;
277         (rune_expecting == rune_input || btc_expecting == btc_input)
278             .then(|| ())
279             .ok_or(ExchangeError::InvalidSignPsbtArgs(
280                 "inputs mismatch with pre_add_liquidity".to_string(),
281             ))?;
282         // calculate the pool state
283         let sats_input: u64 = btc_input
284             .value
285             .try_into()
286             .map_err(|_| ExchangeError::Overflow)?;
287         let (btc_output, rune_output) = pool_utxo
288             .as_ref()
289             .map(|u| {
290                 (
291                     u.satoshis.checked_add(sats_input),
292                     u.balance.value.checked_add(rune_input.value),
293                 )
294             })
295             .unwrap_or_default();
296         let (btc_output, rune_output) = (
297             btc_output.ok_or(ExchangeError::Overflow)?,
298             rune_output.ok_or(ExchangeError::Overflow)?,
299         );
300         let user_k = btc_input
301             .value
302             .checked_mul(rune_input.value)
303             .ok_or(ExchangeError::Overflow)?;
304         let user_share = crate::sqrt(user_k);
305         let pool_output = Utxo::try_from(
306             pool_new_outpoint,
307             CoinBalance {
308                 value: rune_output,
309                 id: rune_input.id,
310             },
311             btc_output,
312         )?;
313         state.utxo = Some(pool_output);
314         state
315             .lp
316             .entry(initiator)
317             .and_modify(|lp| *lp += user_share)
318             .or_insert(user_share);
319         state.k = state.rune_supply() * state.btc_supply() as u128;
320         state.nonce += 1;
321         state.id = Some(txid);
322         Ok((state, pool_utxo))
323     }
```

Listing 2.1: pool.rs

Impact Liquidity can not be correctly added into empty pools.

Suggestion Revise the logic to handle liquidity addition to empty pools correctly, ensuring the function implementation aligns with the design.

2.1.2 PSBT signing fails due to missing POOL_ADDR registration

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `sign_psbt()` function, which is responsible for signing a `PSBT`, fails due to an issue with `POOL_ADDR` not being registered during the creation of a pool. Specifically, the function relies on `POOL_ADDR` to retrieve the corresponding `pool_key` based on the given `pool_address`. However, when a new pool is created, the `create_empty_pool()` function does not register the `pool_address` and its corresponding `pool_key` in `POOL_ADDR`, leaving it empty. As a result, the function `sign_psbt()` is unable to locate the required key, preventing it from signing any `PSBT`.

```
 95   pub async fn create(rune_id: CoinId) -> Result<Pubkey, ExchangeError> {
 96       match crate::with_pool_name(&rune_id) {
 97           Some(pubkey) => crate::with_pool(&pubkey, |pool| {
 98               pool.as_ref()
 99                   .filter(|p| p.states.is_empty())
100                   .map(|p| p.pubkey.clone())
101                   .ok_or(ExchangeError::PoolAlreadyExists)
102           }),
103           None => {
104               let untweaked_pubkey = crate::request_schnorr_key("key_1", rune_id.to_bytes()).
                      await?;
105               let principal = Principal::from_str(crate::RUNE_INDEXER_CANISTER).unwrap();
106               let indexer = RuneIndexer(principal);
107               let (entry,): (Option<RuneEntry>,) = indexer
108                   .get_rune_by_id(rune_id.to_string())
109                   .await
110                   .inspect_err(|e| log!(ERROR, "Error fetching rune indexer: {}", e.1))
111                   .map_err(|_| ExchangeError::FetchRuneIndexerError)?;
112               let name = entry
113                   .map(|e| e.spaced_rune)
114                   .ok_or(ExchangeError::InvalidRuneId)?;
115               let meta = CoinMeta {
116                   id: rune_id,
117                   symbol: name,
118                   min_amount: 1,
119               };
120               crate::create_empty_pool(meta, untweaked_pubkey.clone())?;
121               Ok(untweaked_pubkey)
122           }
123       }
124   }
```

Listing 2.2: canister.rs

```
246   pub(crate) fn create_empty_pool(meta: CoinMeta, untweaked: Pubkey) -> Result<(), ExchangeError
          > {
247       if has_pool(&meta.id) {
248           return Err(ExchangeError::PoolAlreadyExists);
249       }
250       let id = meta.id;
251       let pool =
252           LiquidityPool::new_empty(meta, DEFAULT_FEE_RATE, DEFAULT_BURN_RATE, untweaked.clone())
253               .expect("didn't set fee rate");
254       POOL_TOKENS.with_borrow_mut(|l| {
255           l.insert(id, untweaked.clone());
256           POOLS.with_borrow_mut(|p| {
257               p.insert(untweaked, pool);
258           });
259       });
260       Ok(())
261   }
```

Listing 2.3: lib.rs

Impact This issue effectively blocks all `PSBT` signing operations, preventing transactions from being processed.

Suggestion Revise the logic to add the `pool_address` and its corresponding `pool_key` into the `POOL_ADDR` variable when creating a new pool.

2.1.3 Incorrect rounding direction in swap and withdraw calculations

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The functions `available_to_swap()` and `available_to_withdraw()` both involve division operations, which introduce precision loss. Currently, the rounding direction favors the user, which can lead to an overestimation of the amount of tokens received during swaps and an underestimation of the number of `LP` shares that must be burned during withdrawals.

```
596   pub(crate) fn available_to_swap(
597       &self,
598       taker: CoinBalance,
599   ) -> Result<(CoinBalance, u64, u64), ExchangeError> {
600       let btc_meta = CoinMeta::btc();
601       (taker.id == self.meta.id || taker.id == CoinId::btc())
602           .then(|| ())
603           .ok_or(ExchangeError::InvalidPool)?;
604       let recent_state = self.states.last().ok_or(ExchangeError::EmptyPool)?;
605       let btc_supply = recent_state.btc_supply();
606       let rune_supply = recent_state.rune_supply();
607       (btc_supply != 0 && rune_supply != 0)
608           .then(|| ())
609           .ok_or(ExchangeError::EmptyPool)?;
610       let k = recent_state.btc_supply() as u128 * recent_state.rune_supply();
```

```
611         if taker.id == CoinId::btc() {
612             // btc -> rune
613             let input_btc: u64 = taker.value.try_into().expect("BTC amount overflow");
614             let (input_amount, fee, burn) =
615                 Self::charge_fee(input_btc, self.fee_rate, self.burn_rate);
616             let rune_remains = btc_supply
617                 .checked_add(input_amount)
618                 .and_then(|sum| k.checked_div(sum as u128))
619                 .ok_or(ExchangeError::Overflow)?;
620             (rune_remains >= self.meta.min_amount)
621                 .then(|| ())
622                 .ok_or(ExchangeError::EmptyPool)?;
623             let offer = rune_supply - rune_remains;
624             Ok((
625                 CoinBalance {
626                     value: offer,
627                     id: self.meta.id,
628                 },
629                 fee,
630                 burn,
631             ))
632         } else {
633             // rune -> btc
634             let btc_remains = rune_supply
635                 .checked_add(taker.value)
636                 .and_then(|sum| k.checked_div(sum))
637                 .ok_or(ExchangeError::Overflow)?;
638             // we must ensure that utxo of pool should be >= 546 to hold the dust
639             (btc_remains + recent_state.incomes as u128 >= btc_meta.min_amount)
640                 .then(|| ())
641                 .ok_or(ExchangeError::EmptyPool)?;
642             let btc_remains: u64 = btc_remains.try_into().expect("BTC amount overflow");
643             let pre_charge = btc_supply - btc_remains;
644             let (offer, fee, burn) = Self::charge_fee(pre_charge, self.fee_rate, self.burn_rate);
645             Ok((
646                 CoinBalance {
647                     id: btc_meta.id,
648                     value: offer as u128,
649                 },
650                 fee,
651                 burn,
652             ))
653         }
654     }
```

Listing 2.4: pool.rs

```
409     pub(crate) fn available_to_withdraw(
410         &self,
411         pubkey_hash: impl AsRef<str>,
412         btc_delta: u128,
413     ) -> Result<(u64, CoinBalance, u128), ExchangeError> {
414         let recent_state = self.states.last().ok_or(ExchangeError::EmptyPool)?;
```

```
415         let lp = recent_state.lp(pubkey_hash.as_ref());
416         (lp != 0).then(|| ()).ok_or(ExchangeError::LpNotFound)?;
417 
418         // global
419         let sqrt_k = crate::sqrt(recent_state.btc_supply() as u128 * recent_state.rune_supply());
420         let btc_supply = recent_state.btc_supply();
421         let rune_supply = recent_state.rune_supply();
422 
423         let mut btc_delta = btc_delta;
424         let part_k = btc_delta
425             .checked_mul(sqrt_k)
426             .and_then(|m| m.checked_div(btc_supply as u128))
427             .ok_or(ExchangeError::InsufficientFunds)?;
428         (part_k <= lp)
429             .then(|| ())
430             .ok_or(ExchangeError::InsufficientFunds)?;
431 
432         let mut rune_delta = part_k
433             .checked_mul(rune_supply)
434             .and_then(|m| m.checked_div(sqrt_k))
435             .ok_or(ExchangeError::EmptyPool)?;
436         let btc_remains = recent_state
437             .satoshis()
438             .checked_sub(btc_delta as u64)
439             .ok_or(ExchangeError::EmptyPool)?;
440         let mut k = 0u128;
441         if btc_remains < CoinMeta::btc().min_amount as u64 {
442             // reward the dust to the last valid lp
443             btc_delta += btc_remains as u128;
444             rune_delta = rune_supply;
445         } else {
446             let btc_total = lp
447                 .checked_mul(btc_supply as u128)
448                 .and_then(|r| r.checked_div(sqrt_k))
449                 .ok_or(ExchangeError::InsufficientFunds)?;
450             let rune_total = lp
451                 .checked_mul(rune_supply)
452                 .and_then(|m| m.checked_div(sqrt_k))
453                 .ok_or(ExchangeError::InsufficientFunds)?;
454             let btc_user_remain = btc_total
455                 .checked_sub(btc_delta)
456                 .ok_or(ExchangeError::InsufficientFunds)?;
457             let rune_user_remain = rune_total
458                 .checked_sub(rune_delta)
459                 .ok_or(ExchangeError::InsufficientFunds)?;
460             let new_user_share = btc_user_remain
461                 .checked_mul(rune_user_remain)
462                 .ok_or(ExchangeError::Overflow)?;
463             k = crate::sqrt(new_user_share);
464         }
465         Ok((
466             btc_delta.try_into().map_err(|_| ExchangeError::Overflow)?,
467             CoinBalance {
```

```
468                 id: self.meta.id,
469                 value: rune_delta,
470             },
471             k,
472         ))
473     }
```

Listing 2.5: pool.rs

Impact Due to the incorrect rounding direction in the calculations, users may receive more tokens than expected during withdrawals or swaps.

Suggestion Revise the logic to ensure that the rounding direction benefits the protocol.

Feedback from the Project The precision loss is within the acceptable range.

2.1.4 Underestimation of user's share amount due to precision loss

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The current implementation incorrectly calculates liquidity shares when users add or withdraw liquidity, leading to systematic underestimation over time. Instead of maintaining a total `LP` share supply and minting shares based on the proportion of added liquidity, the system calculates shares using a square root function (i.e., function `sqrt()`). This approach introduces precision loss, as decimal truncation causes users to receive fewer `LP` shares than they should when adding liquidity. Similarly, when withdrawing, the system first determines the amount of `LP` shares needed to withdraw a given asset, then recalculates the user's remaining share using another square root operation, further compounding the loss. As a result, users gradually lose liquidity share with each transaction, making it impossible to fully withdraw funds over time.

```
226     pub(crate) fn validate_adding_liquidity(
227         &self,
228         txid: Txid,
229         nonce: u64,
230         pool_utxo_spend: Vec<String>,
231         pool_utxo_receive: Vec<String>,
232         input_coins: Vec<InputCoin>,
233         output_coins: Vec<OutputCoin>,
234         initiator: String,
235     ) -> Result<(PoolState, Option<Utxo>), ExchangeError> {
236         (input_coins.len() == 2 && output_coins.is_empty())
237             .then(|| ())
238             .ok_or(ExchangeError::InvalidSignPsbtArgs(
239                 "invalid input/output_coins, add_liquidity requires 2 inputs and 0 output"
240                     .to_string(),
241             ))?;
242         let x = input_coins[0].coin.clone();
243         let y = input_coins[1].coin.clone();
244         let mut state = self.states.last().cloned().unwrap_or_default();
```

```
        // check nonce matches
        (state.nonce == nonce)
            .then(|| ())
            .ok_or(ExchangeError::PoolStateExpired(state.nonce))?;
        // check prev_outpoint matches
        let pool_utxo = state.utxo.clone();
        (pool_utxo.as_ref().map(|u| u.outpoint()).as_ref() == pool_utxo_spend.last())
            .then(|| ())
            .ok_or(ExchangeError::InvalidSignPsbtArgs(
                "pool_utxo_spend/pool state mismatch".to_string(),
            ))?;
        // check output exists
        let pool_new_outpoint = pool_utxo_receive.last().map(|s| s.clone()).ok_or(
            ExchangeError::InvalidSignPsbtArgs("pool_utxo_receive not found".to_string()),
        )?;
        // check input coins
        let (btc_input, rune_input) = if x.id == CoinId::btc() && y.id != CoinId::btc() {
            Ok((x, y))
        } else if x.id != CoinId::btc() && y.id == CoinId::btc() {
            Ok((y, x))
        } else {
            Err(ExchangeError::InvalidSignPsbtArgs(
                "Invalid inputs: requires 2 different input coins".to_string(),
            ))
        }?;
        // check minial liquidity
        (btc_input.value >= MIN_BTC_VALUE as u128)
            .then(|| ())
            .ok_or(ExchangeError::TooSmallFunds)?;
        // y = f(x), x' = f(y'); => x == x' || y == y'
        let rune_expecting = self.liquidity_should_add(btc_input)?;
        let btc_expecting = self.liquidity_should_add(rune_input)?;
        (rune_expecting == rune_input || btc_expecting == btc_input)
            .then(|| ())
            .ok_or(ExchangeError::InvalidSignPsbtArgs(
                "inputs mismatch with pre_add_liquidity".to_string(),
            ))?;
        // calculate the pool state
        let sats_input: u64 = btc_input
            .value
            .try_into()
            .map_err(|_| ExchangeError::Overflow)?;
        let (btc_output, rune_output) = pool_utxo
            .as_ref()
            .map(|u| {
                (
                    u.satoshis.checked_add(sats_input),
                    u.balance.value.checked_add(rune_input.value),
                )
            })
            .unwrap_or_default();
        let (btc_output, rune_output) = (
            btc_output.ok_or(ExchangeError::Overflow)?,
```

```
298             rune_output.ok_or(ExchangeError::Overflow)?,
299         );
300         let user_k = btc_input
301             .value
302             .checked_mul(rune_input.value)
303             .ok_or(ExchangeError::Overflow)?;
304         let user_share = crate::sqrt(user_k);
305         let pool_output = Utxo::try_from(
306             pool_new_outpoint,
307             CoinBalance {
308                 value: rune_output,
309                 id: rune_input.id,
310             },
311             btc_output,
312         )?;
313         state.utxo = Some(pool_output);
314         state
315             .lp
316             .entry(initiator)
317             .and_modify(|lp| *lp += user_share)
318             .or_insert(user_share);
319         state.k = state.rune_supply() * state.btc_supply() as u128;
320         state.nonce += 1;
321         state.id = Some(txid);
322         Ok((state, pool_utxo))
323     }
```

Listing 2.6: pool.rs

```
409     pub(crate) fn available_to_withdraw(
410         &self,
411         pubkey_hash: impl AsRef<str>,
412         btc_delta: u128,
413     ) -> Result<(u64, CoinBalance, u128), ExchangeError> {
414         let recent_state = self.states.last().ok_or(ExchangeError::EmptyPool)?;
415         let lp = recent_state.lp(pubkey_hash.as_ref());
416         (lp != 0).then(|| ()).ok_or(ExchangeError::LpNotFound)?;
417 
418         // global
419         let sqrt_k = crate::sqrt(recent_state.btc_supply() as u128 * recent_state.rune_supply());
420         let btc_supply = recent_state.btc_supply();
421         let rune_supply = recent_state.rune_supply();
422 
423         let mut btc_delta = btc_delta;
424         let part_k = btc_delta
425             .checked_mul(sqrt_k)
426             .and_then(|m| m.checked_div(btc_supply as u128))
427             .ok_or(ExchangeError::InsufficientFunds)?;
428         (part_k <= lp)
429             .then(|| ())
430             .ok_or(ExchangeError::InsufficientFunds)?;
431 
432         let mut rune_delta = part_k
```

```
433             .checked_mul(rune_supply)
434             .and_then(|m| m.checked_div(sqrt_k))
435             .ok_or(ExchangeError::EmptyPool)?;
436         let btc_remains = recent_state
437             .satoshis()
438             .checked_sub(btc_delta as u64)
439             .ok_or(ExchangeError::EmptyPool)?;
440         let mut k = 0u128;
441         if btc_remains < CoinMeta::btc().min_amount as u64 {
442             // reward the dust to the last valid lp
443             btc_delta += btc_remains as u128;
444             rune_delta = rune_supply;
445         } else {
446             let btc_total = lp
447                 .checked_mul(btc_supply as u128)
448                 .and_then(|r| r.checked_div(sqrt_k))
449                 .ok_or(ExchangeError::InsufficientFunds)?;
450             let rune_total = lp
451                 .checked_mul(rune_supply)
452                 .and_then(|m| m.checked_div(sqrt_k))
453                 .ok_or(ExchangeError::InsufficientFunds)?;
454             let btc_user_remain = btc_total
455                 .checked_sub(btc_delta)
456                 .ok_or(ExchangeError::InsufficientFunds)?;
457             let rune_user_remain = rune_total
458                 .checked_sub(rune_delta)
459                 .ok_or(ExchangeError::InsufficientFunds)?;
460             let new_user_share = btc_user_remain
461                 .checked_mul(rune_user_remain)
462                 .ok_or(ExchangeError::Overflow)?;
463             k = crate::sqrt(new_user_share);
464         }
465         Ok((
466             btc_delta.try_into().map_err(|_| ExchangeError::Overflow)?,
467             CoinBalance {
468                 id: self.meta.id,
469                 value: rune_delta,
470             },
471             k,
472         ))
473     }
```

Listing 2.7: pool.rs

Impact The user's share amount is underestimated when adding and withdrawing liquidity.

Suggestion To fix this, the protocol should adopt a total `LP` share supply model, ensuring shares are distributed proportionally rather than relying on square root calculations that introduce rounding errors.

2.1.5 Potential cycle drain due to lack of access control in function `create()`

Severity Medium

Status Confirmed

Introduced by Version 1

Description The `create()` function allows users to create liquidity pools based on a given `rune_id`. If the specified pool does not exist, the function makes an inter-canister invocation to the `Rune Indexer Canister` via the function `get_rune_by_id()`, fetching rune information before creating a new pool. Since inter-canister invokes on `ICP` require the invoking `canister` to pay for message transmission, a malicious user can repeatedly invoke the function `create()`, forcing the canister to issue expensive external calls. Without access control, this mechanism can be exploited to drain the `canister`'s `cycle` balance, eventually rendering the protocol unusable.

```
 95   pub async fn create(rune_id: CoinId) -> Result<Pubkey, ExchangeError> {
 96       match crate::with_pool_name(&rune_id) {
 97           Some(pubkey) => crate::with_pool(&pubkey, |pool| {
 98               pool.as_ref()
 99                   .filter(|p| p.states.is_empty())
100                   .map(|p| p.pubkey.clone())
101                   .ok_or(ExchangeError::PoolAlreadyExists)
102           }),
103           None => {
104               let untweaked_pubkey = crate::request_schnorr_key("key_1", rune_id.to_bytes()).
                      await?;
105               let principal = Principal::from_str(crate::RUNE_INDEXER_CANISTER).unwrap();
106               let indexer = RuneIndexer(principal);
107               let (entry,): (Option<RuneEntry>,) = indexer
108                   .get_rune_by_id(rune_id.to_string())
109                   .await
110                   .inspect_err(|e| log!(ERROR, "Error fetching rune indexer: {}", e.1))
111                   .map_err(|_| ExchangeError::FetchRuneIndexerError)?;
112               let name = entry
113                   .map(|e| e.spaced_rune)
114                   .ok_or(ExchangeError::InvalidRuneId)?;
115               let meta = CoinMeta {
116                   id: rune_id,
117                   symbol: name,
118                   min_amount: 1,
119               };
120               crate::create_empty_pool(meta, untweaked_pubkey.clone())?;
121               Ok(untweaked_pubkey)
122           }
123       }
124   }
```

Listing 2.8: canister.rs

Impact A malicious user may drain the `canister`'s `cycle` balance, preventing other users from normally using the protocol.

Suggestion Add access control to ensure that the function `create()` is not abused.

2.1.6 Failure of extracting protocol fees in function `available_to_extract()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `available_to_extract()` function checks whether the protocol can extract the accumulated fee (`incomes`) by performing three conditions checks. However, these checks contain logical errors that can prevent the protocol from extracting fees even when sufficient income has been accrued.

Condition 1: The function requires `btc_supply >= CoinMeta::btc().min_amount` (`546 satoshis`), which ensures that the pool has a minimum `BTC` amount. However, if fees have been accumulated through swaps but liquidity is later withdrawn, reducing `btc_supply` below this threshold, the function will fail to extract fees, even if `recent_state.incomes` is sufficient.

Condition 2: The function only checks if `recent_state.incomes > 0`, but since extracting fees involves creating a `BTC UTXO`, the correct threshold should be `recent_state.incomes > 546` to ensure the fee can actually be transferred.

Condition 3: The third condition, `btc_supply - recent_state.incomes >= CoinMeta::btc().min_amount`, is redundant because `btc_supply` is already defined as `utxo.satoshis - self.incomes`. Subtracting `recent_state.incomes` again effectively applies the same constraint twice.

```
77   pub fn btc_supply(&self) -> u64 {
78       self.utxo
79           .as_ref()
80           .map(|utxo| utxo.satoshis - self.incomes)
81           .unwrap_or_default()
82   }
```

Listing 2.9: pool.rs

```
325  pub(crate) fn available_to_extract(&self) -> Result<u64, ExchangeError> {
326      let recent_state = self.states.last().ok_or(ExchangeError::EmptyPool)?;
327      let btc_supply = recent_state.btc_supply();
328      // TODO improve this
329      (btc_supply >= CoinMeta::btc().min_amount as u64
330          && recent_state.incomes > 0
331          && btc_supply - recent_state.incomes >= CoinMeta::btc().min_amount as u64)
332          .then(|| ())
333          .ok_or(ExchangeError::InvalidLiquidity)?;
334      Ok(recent_state.incomes)
335  }
```

Listing 2.10: pool.rs

Impact The `incomes` might not be extracted properly.

Suggestion Modify the above three checks to ensure that the protocol fee (`incomes`) can be properly extracted when it reaches the minimum amount.

2.1.7 Incorrect check in function `available_to_swap()`

Severity Medium

Status Fixed in `Version 2`

Introduced by Version 1

Description The function `available_to_swap()` calculates the output token quantity based on the user's input and the latest pool state. When swapping `Rune` for `BTC`, the pool's `BTC` balance decreases. To ensure the remaining `UTXO` in the pool stays above the minimum transfer amount of `546`, the function currently applies the check: `btc_remains + recent_state.incomes as u128 >= btc_meta.min_amount`. However, this check is incorrect for two reasons:

- The pool's `UTXO` and `incomes` should be handled separately. Including `incomes` in this check can lead to miscalculations, as it represents protocol fees rather than the liquidity available to swap.
- The current `btc_remains` value represents the remaining `UTXO` after the swap, computed using the constant product formula. However, this calculation does not include the `swap fee` for liquidity providers, which should be included to ensure an accurate check against the minimum `UTXO` requirement.

```
596   pub(crate) fn available_to_swap(
597       &self,
598       taker: CoinBalance,
599   ) -> Result<(CoinBalance, u64, u64), ExchangeError> {
600       let btc_meta = CoinMeta::btc();
601       (taker.id == self.meta.id || taker.id == CoinId::btc())
602           .then(|| ())
603           .ok_or(ExchangeError::InvalidPool)?;
604       let recent_state = self.states.last().ok_or(ExchangeError::EmptyPool)?;
605       let btc_supply = recent_state.btc_supply();
606       let rune_supply = recent_state.rune_supply();
607       (btc_supply != 0 && rune_supply != 0)
608           .then(|| ())
609           .ok_or(ExchangeError::EmptyPool)?;
610       let k = recent_state.btc_supply() as u128 * recent_state.rune_supply();
611       if taker.id == CoinId::btc() {
612           // btc -> rune
613           let input_btc: u64 = taker.value.try_into().expect("BTC amount overflow");
614           let (input_amount, fee, burn) =
615               Self::charge_fee(input_btc, self.fee_rate, self.burn_rate);
616           let rune_remains = btc_supply
617               .checked_add(input_amount)
618               .and_then(|sum| k.checked_div(sum as u128))
619               .ok_or(ExchangeError::Overflow)?;
620           (rune_remains >= self.meta.min_amount)
621               .then(|| ())
622               .ok_or(ExchangeError::EmptyPool)?;
623           let offer = rune_supply - rune_remains;
624           Ok((
625               CoinBalance {
626                   value: offer,
627                   id: self.meta.id,
628               },
629               fee,
630               burn,
631           ))
```

```
632         } else {
633             // rune -> btc
634             let btc_remains = rune_supply
635                 .checked_add(taker.value)
636                 .and_then(|sum| k.checked_div(sum))
637                 .ok_or(ExchangeError::Overflow)?;
638             // we must ensure that utxo of pool should be >= 546 to hold the dust
639             (btc_remains + recent_state.incomes as u128 >= btc_meta.min_amount)
640                 .then(|| ())
641                 .ok_or(ExchangeError::EmptyPool)?;
642             let btc_remains: u64 = btc_remains.try_into().expect("BTC amount overflow");
643             let pre_charge = btc_supply - btc_remains;
644             let (offer, fee, burn) = Self::charge_fee(pre_charge, self.fee_rate, self.burn_rate);
645             Ok((
646                 CoinBalance {
647                     id: btc_meta.id,
648                     value: offer as u128,
649                 },
650                 fee,
651                 burn,
652             ))
653         }
654     }
```

Listing 2.11: pool.rs

Impact The incorrect `UTXO` calculation may lead to an inaccurate assessment of the pool's available liquidity after a swap.

Suggestion Modify the logic to properly separate `recent_state.incomes` from the `UTXO` check and ensure that transaction fees are accounted for. The revised check should explicitly verify that the post-swap pool `UTXO` remains at or above the `546-satoshi` threshold.

2.2 Additional Recommendation

2.2.1 Redundant code

Status Confirmed

Introduced by `Version 1`

Description The `new_empty()` function imposes constraints on `fee_rate` and `burn_rate` by requiring `fee_rate <= 1_000_000` and `burn_rate <= 1_000_000`. However, `new_empty()` is only invoked in function `create_empty_pool()`, which passes the constants `DEFAULT_FEE_RATE` (7000) and `DEFAULT_BURN_RATE` (2000) as arguments—both of which are significantly below the 1_000_000 threshold. Therefore, these constraints are redundant.

```
127     pub fn new_empty(
128         meta: CoinMeta,
129         fee_rate: u64,
130         burn_rate: u64,
131         untweaked: Pubkey,
```

```
132   ) -> Option<Self> {
133       (fee_rate <= 1_000_000).then(|| ())?;
134       (burn_rate <= 1_000_000).then(|| ())?;
135       let tweaked = crate::tweak_pubkey_with_empty(untweaked.clone());
136       let key = ree_types::bitcoin::key::TweakedPublicKey::dangerous_assume_tweaked(
137           tweaked.to_x_only_public_key(),
138       );
139       let addr = Address::p2tr_tweaked(key, Network::Bitcoin).to_string();
140       Some(Self {
141           states: vec![],
142           fee_rate,
143           burn_rate,
144           meta,
145           pubkey: untweaked,
146           tweaked,
147           addr,
148       })
149   }
```

Listing 2.12: pool.rs

```
246   pub(crate) fn create_empty_pool(meta: CoinMeta, untweaked: Pubkey) -> Result<(), ExchangeError
          > {
247       if has_pool(&meta.id) {
248           return Err(ExchangeError::PoolAlreadyExists);
249       }
250       let id = meta.id;
251       let pool =
252           LiquidityPool::new_empty(meta, DEFAULT_FEE_RATE, DEFAULT_BURN_RATE, untweaked.clone())
253               .expect("didn't set fee rate");
254       POOL_TOKENS.with_borrow_mut(|l| {
255           l.insert(id, untweaked.clone());
256           POOLS.with_borrow_mut(|p| {
257               p.insert(untweaked, pool);
258           });
259       });
260       Ok(())
261   }
```

Listing 2.13: lib.rs

Suggestion Remove the redundant code..

2.2.2 Add crate visibility modifier to function `validate_extract_fee()`

Status Confirmed

Introduced by `Version 1`

Description The `validate_extract_fee()` function is missing the `crate` visibility modifier, which is inconsistent with other validation functions in the codebase. To ensure uniformity and maintain consistency in the function's visibility, it is recommended to add the `crate` keyword to this function.

```rust
pub fn validate_extract_fee(
    &self,
    txid: Txid,
    nonce: u64,
    pool_utxo_spend: Vec<String>,
    pool_utxo_receive: Vec<String>,
    input_coins: Vec<InputCoin>,
    output_coins: Vec<OutputCoin>,
) -> Result<(PoolState, Utxo), ExchangeError> {
    (input_coins.is_empty() && output_coins.len() == 1)
        .then(|| ())
        .ok_or(ExchangeError::InvalidSignPsbtArgs(
            "invalid input/output coins, extract fee requires 0 input and 1 output".to_string(),
        ))?;
    let output = output_coins.first().clone().expect("checked;qed");
    let fee_collector = crate::p2tr_untweaked(&crate::get_fee_collector());
    (output.coin.id == CoinMeta::btc().id && output.to == fee_collector)
        .then(|| ())
        .ok_or(ExchangeError::InvalidSignPsbtArgs(format!(
            "invalid output coin, extract fee requires 1 output of BTC to {}",
            fee_collector
        )))?;
    let mut state = self
        .states
        .last()
        .cloned()
        .ok_or(ExchangeError::EmptyPool)?;
    // check nonce
    (state.nonce == nonce)
        .then(|| ())
        .ok_or(ExchangeError::PoolStateExpired(state.nonce))?;
    let prev_outpoint =
        pool_utxo_spend
            .last()
            .map(|s| s.clone())
            .ok_or(ExchangeError::InvalidSignPsbtArgs(
                "pool_utxo_spend not found".to_string(),
            ))?;
    let prev_utxo = state.utxo.clone().ok_or(ExchangeError::EmptyPool)?;
    (prev_outpoint == prev_utxo.outpoint()).then(|| ()).ok_or(
        ExchangeError::InvalidSignPsbtArgs("pool_utxo_spend/pool state mismatch".to_string()),
    )?;
    let btc_delta = self.available_to_extract()?;
    (output.coin.value == btc_delta as u128).then(|| ()).ok_or(
        ExchangeError::InvalidSignPsbtArgs(
            "invalid output coin, extract fee requires 1 output of BTC with correct value"
                .to_string(),
        ),
    )?;
    let pool_output = if btc_delta == prev_utxo.satoshis {
        None
```

```
388         } else {
389             Some(Utxo::try_from(
390                 pool_utxo_receive
391                     .last()
392                     .ok_or(ExchangeError::InvalidSignPsbtArgs(
393                         "pool_utxo_receive not found".to_string(),
394                     ))?,
395                 CoinBalance {
396                     id: self.base_id(),
397                     value: prev_utxo.balance.value,
398                 },
399                 prev_utxo.satoshis - btc_delta,
400             )?)
401         };
402         state.utxo = pool_output;
403         state.incomes = 0;
404         state.nonce += 1;
405         state.id = Some(txid);
406         Ok((state, prev_utxo))
407     }
```

Listing 2.14: pool.rs

Suggestion Add the `crate` keyword to `validate_extract_fee()` function.

2.2.3 Comments typo correction

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There are typo errors in the comments within the functions `validate_withdrawing_liquidity()` and `validate_swap()`. Specifically, "`chech minial sats`" and "`chech params`" should be corrected to "`check minimal sats`" and "`check params`", respectively.

```
475     pub(crate) fn validate_withdrawing_liquidity(
476         &self,
477         txid: Txid,
478         nonce: u64,
479         pool_utxo_spend: Vec<String>,
480         pool_utxo_receive: Vec<String>,
481         input_coins: Vec<InputCoin>,
482         output_coins: Vec<OutputCoin>,
483         initiator: String,
484     ) -> Result<(PoolState, Utxo), ExchangeError> {
485         (input_coins.is_empty() && output_coins.len() == 2)
486             .then(|| ())
487             .ok_or(ExchangeError::InvalidSignPsbtArgs(
488                 "invalid input/output_coins, withdraw_liquidity requires 0 input and 2 outputs"
489                     .to_string(),
490             ))?;
491         let x = output_coins[0].coin.clone();
492         let y = output_coins[1].coin.clone();
493         let (btc_output, rune_output) = if x.id == CoinId::btc() && y.id != CoinId::btc() {
```

```
494             Ok((x, y))
495         } else if x.id != CoinId::btc() && y.id == CoinId::btc() {
496             Ok((y, x))
497         } else {
498             Err(ExchangeError::InvalidSignPsbtArgs(
499                 "Invalid outputs: requires 2 different output coins".to_string(),
500             ))
501         }?;
502         let pool_prev_outpoint =
503             pool_utxo_spend
504                 .last()
505                 .map(|s| s.clone())
506                 .ok_or(ExchangeError::InvalidSignPsbtArgs(
507                     "pool_utxo_spend not found".to_string(),
508                 ))?;
509         let mut state = self.states.last().ok_or(ExchangeError::EmptyPool)?.clone();
510         // check nonce
511         (state.nonce == nonce)
512             .then(|| ())
513             .ok_or(ExchangeError::PoolStateExpired(state.nonce))?;
514         // check prev state
515         let prev_utxo = state.utxo.clone().ok_or(ExchangeError::EmptyPool)?;
516         (prev_utxo.outpoint() == pool_prev_outpoint)
517             .then(|| ())
518             .ok_or(ExchangeError::InvalidSignPsbtArgs(
519                 "pool_utxo_spend/pool_state don't match".to_string(),
520             ))?;
521         // chech minial sats
522         (btc_output.value >= MIN_BTC_VALUE as u128)
523             .then(|| ())
524             .ok_or(ExchangeError::TooSmallFunds)?;
525         // chech params
526         let k = state.rune_supply() * state.btc_supply() as u128;
527         let (btc_expecting, rune_expecting, new_share) =
528             // a user wants to withdraw all(including incomes), we must check its share is 100%
529             if btc_output.value == state.satoshis() as u128 {
530                 let lp = state.lp(&initiator);
531                 let real_btc = lp
532                     .checked_mul(state.btc_supply() as u128)
533                     .and_then(|share| share.checked_div(crate::sqrt(k)))
534                     .ok_or(ExchangeError::Overflow)?;
535                 self.available_to_withdraw(&initiator, real_btc)?
536             } else {
537                 self.available_to_withdraw(&initiator, btc_output.value)?
538             };
539         let btc_output: u64 = btc_output
540             .value
541             .try_into()
542             .map_err(|_| ExchangeError::Overflow)?;
543         (rune_expecting == rune_output && btc_expecting == btc_output)
544             .then(|| ())
545             .ok_or(ExchangeError::InvalidSignPsbtArgs(
546                 "inputs mismatch with pre_withdraw_liquidity".to_string(),
```

```
547             ))?;
548         let (pool_btc_output, pool_rune_output) = (
549             prev_utxo
550                 .satoshis
551                 .checked_sub(btc_output)
552                 .ok_or(ExchangeError::Overflow)?,
553             prev_utxo
554                 .balance
555                 .value
556                 .checked_sub(rune_output.value)
557                 .ok_or(ExchangeError::Overflow)?,
558         );
559         let pool_should_receive = pool_btc_output != 0 || pool_rune_output != 0;
560         let new_utxo = if pool_should_receive {
561             Some(Utxo::try_from(
562                 pool_utxo_receive
563                     .last()
564                     .ok_or(ExchangeError::InvalidSignPsbtArgs(
565                         "pool_utxo_receive not found".to_string(),
566                     ))?,
567                 CoinBalance {
568                     id: rune_output.id,
569                     value: pool_rune_output,
570                 },
571                 pool_btc_output,
572             )?)
573         } else {
574             None
575         };
576         state.utxo = new_utxo;
577         state.k = state.rune_supply() * state.btc_supply() as u128;
578         if state.utxo.is_none() {
579             state.incomes = 0;
580             state.lp.clear();
581         } else {
582             if new_share != 0 {
583                 state.lp.insert(initiator, new_share);
584             } else {
585                 state.lp.remove(&initiator);
586             }
587         }
588         state.nonce += 1;
589         state.id = Some(txid);
590         Ok((state, prev_utxo))
591     }
```

Listing 2.15: pool.rs

```
656     pub(crate) fn validate_swap(
657         &self,
658         txid: Txid,
659         nonce: u64,
660         pool_utxo_spend: Vec<String>,
```

```
661         pool_utxo_receive: Vec<String>,
662         input_coins: Vec<InputCoin>,
663         output_coins: Vec<OutputCoin>,
664     ) -> Result<(PoolState, Utxo), ExchangeError> {
665         (input_coins.len() == 1 && output_coins.len() == 1)
666             .then(|| ())
667             .ok_or(ExchangeError::InvalidSignPsbtArgs(
668                 "invalid input/output coins, swap requires 1 input and 1 output".to_string(),
669             ))?;
670         let input = input_coins.first().clone().expect("checked;qed");
671         let output = output_coins.first().clone().expect("checked;qed");
672         let mut state = self
673             .states
674             .last()
675             .cloned()
676             .ok_or(ExchangeError::EmptyPool)?;
677         // check nonce
678         (state.nonce == nonce)
679             .then(|| ())
680             .ok_or(ExchangeError::PoolStateExpired(state.nonce))?;
681         let prev_outpoint =
682             pool_utxo_spend
683                 .last()
684                 .map(|s| s.clone())
685                 .ok_or(ExchangeError::InvalidSignPsbtArgs(
686                     "pool_utxo_spend not found".to_string(),
687                 ))?;
688         let prev_utxo = state.utxo.clone().ok_or(ExchangeError::EmptyPool)?;
689         (prev_outpoint == prev_utxo.outpoint()).then(|| ()).ok_or(
690             ExchangeError::InvalidSignPsbtArgs("pool_utxo_spend/pool state mismatch".to_string()),
691         )?;
692         // chech minimal sats
693         let (offer, _, burn) = self.available_to_swap(input.coin)?;
694         let (btc_output, rune_output) = if input.coin.id == CoinId::btc() {
695             let input_btc: u64 = input
696                 .coin
697                 .value
698                 .try_into()
699                 .map_err(|_| ExchangeError::Overflow)?;
700             (input_btc >= MIN_BTC_VALUE)
701                 .then(|| ())
702                 .ok_or(ExchangeError::TooSmallFunds)?;
703             // assume the user inputs were valid
704             (
705                 prev_utxo.satoshis.checked_add(input_btc),
706                 prev_utxo.balance.value.checked_sub(offer.value),
707             )
708         } else {
709             let output_btc: u64 = offer
710                 .value
711                 .try_into()
712                 .map_err(|_| ExchangeError::Overflow)?;
713             (output_btc >= MIN_BTC_VALUE)
```

```
714                 .then(|| ())
715                 .ok_or(ExchangeError::TooSmallFunds)?;
716             (
717                 prev_utxo.satoshis.checked_sub(output_btc),
718                 prev_utxo.balance.value.checked_add(input.coin.value),
719             )
720         };
721         // check params
722         (output.coin == offer)
723             .then(|| ())
724             .ok_or(ExchangeError::InvalidSignPsbtArgs(
725                 "inputs mismatch with pre_swap".to_string(),
726             ))?;
727         let (btc_output, rune_output) = (
728             btc_output.ok_or(ExchangeError::Overflow)?,
729             rune_output.ok_or(ExchangeError::Overflow)?,
730         );
731         let pool_output = Utxo::try_from(
732             pool_utxo_receive
733                 .last()
734                 .ok_or(ExchangeError::InvalidSignPsbtArgs(
735                     "pool_utxo_receive not found".to_string(),
736                 ))?,
737             CoinBalance {
738                 id: self.base_id(),
739                 value: rune_output,
740             },
741             btc_output,
742         )?;
743         state.utxo = Some(pool_output);
744         state.nonce += 1;
745         state.incomes += burn;
746         state.k = state.rune_supply() * state.btc_supply() as u128;
747         state.id = Some(txid);
748         Ok((state, prev_utxo))
749     }
```

Listing 2.16: pool.rs

Suggestion Correct typo in comments accordingly.

2.3 Note

2.3.1 Potential centralization risk

Description In the current implementation, several privileged roles are set to govern and regulate the system-wide operations (e.g., parameter setting and rollback/finalize transactions). Additionally, the `owner` also has the ability to upgrade the implementation. If the private keys of these privileged roles are lost or maliciously exploited, it could potentially lead to losses for users.

2.3.2 Validated parameter of orchestrator

Description Currently, the `sign_psbt()` function restricts access to only the `orchestrator` role. This function accepts parameters such as `pool_utxo_receive`, `intention_set`, `input_coins`, and `output_coins` to execute subsequent swap or liquidity add/withdraw logic. Notably, these parameters are assumed to be secure and validated by the `orchestrator`. For instance, the `initiator` parameter is obtained from the `intention_set`. The `orchestrator` must implement signature verification to ensure the legitimacy of the `initiator`. Otherwise, this could lead to unauthorized asset withdrawals.

2.3.3 Potential DoS in function `rollback()`

Description The `rollback()` function reverts the pool's state by locating a given `txid` in the state queue and removing that state along with all subsequent ones. This introduces a potential denial-of-service (DoS) risk. If a swap transaction occurs but its `PSBT` fails to commit, all subsequent transactions are rolled back. An attacker could intentionally disrupt the swap process by submitting a valid swap `PSBT`, getting it signed but not yet broadcasted, and then front-running the transaction by transferring their `BTC` to another address with a higher gas fee. The orchestrator should ensure that the attacker cannot profit from this action.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS