Liquidium

Audit Report

Mon Sep 01 2025

contact@bitslab.xyz https://twitter.com/scalebit_

ScaleBit

Liquidium Audit Report

1 Executive Summary

1.1 Project Information

Description	A Bitcoin staking protocol that allows users to stake Bitcoin and receive liquid staking tokens
Type	L2
Auditors	ScaleBit
Timeline	Wed Aug 06 2025 - Mon Sep 01 2025
Languages	Rust
Platform	BTC
Methods	Dependency Check, Fuzzing, Static Analysis, Manual Review
Source Code	https://github.com/Liquidium-Inc/liquidium-staking-canister
Commits	e04b7b70b8b8b5ead630e04889e5e21b5caf10a0 ec34739c59ad6a0c5a894dd16749fcd2342e03fc 4269bc0418ac4b7c0f65fcf54e82d3732268adc9

1.2 Files in Scope

The following are the directories of the original reviewed files.

Directory
https://github.com/Liquidium-Inc/liquidium-staking-canister/src

1.3 Issue Statistic

Item	Count	Fixed	Acknowledged
Total	19	17	2
Informational	1	0	1
Minor	4	4	0
Medium	7	6	1
Major	1	1	0
Critical	6	6	0

1.4 ScaleBit Audit Breakdown

ScaleBit aims to assess repositories for security-related issues, code quality, and compliance with specifications and best practices. Possible issues our team looked for included (but are not limited to):

- Integer overflow/underflow
- Infinite Loop
- Infinite Recursion
- Race Condition
- Traditional Web Vulnerabilities
- Memory Exhaustion Attack
- Disk Space Exhaustion Attack
- Side-channel Attack
- Denial of Service
- Replay Attacks
- Double-spending Attack
- Eclipse Attack
- Sybil Attack
- Eavesdropping Attack
- Business Logic Issues
- Contract Virtual Machine Vulnerabilities
- Coding Style Issues

1.5 Methodology

Our security team adopted **"Dependency Check"**, **"Automated Static Code Analysis"**, **"Fuzz Testing"**, and **"Manual Review"** to conduct a comprehensive security test on the code in a manner closest to real attacks. The main entry points and scope of the security testing are specified in the **"Files in Scope"**, which can be expanded beyond the scope according to actual testing needs. The main types of this security audit include:

(1) Dependency Check

A comprehensive check of the software's dependency libraries was conducted to ensure all external libraries and frameworks are up-to-date and free of known security vulnerabilities.

(2) Automated Static Code Analysis

Static code analysis tools were used to find common programming errors, potential security vulnerabilities, and code patterns that do not conform to best practices.

(3) Fuzz Testing

A large amount of randomly generated data was inputted into the software to try and trigger potential errors and exceptional paths.

(4) Manual Review

The scope of the code is explained in section 1.2.

(5) Audit Process

- Clarify the scope, objectives, and key requirements of the audit.
- Collect related materials such as software documentation, architecture diagrams, and lists of dependency libraries to provide background information for the audit.
- Use automated tools to generate a list of the software's dependency libraries and employ professional tools to scan these libraries for security vulnerabilities, identifying outdated or known vulnerable dependencies.
- Select and configure automated static analysis tools suitable for the project, perform automated scans to identify security vulnerabilities, non-standard coding, and

potential risk points in the code. Evaluate the scanning results to determine which findings require further manual review.

- Design a series of fuzz testing cases aimed at testing the software's ability to handle exceptional data inputs. Analyze the issues found during the testing to determine the defects that need to be fixed.
- Based on the results of the preliminary automated analysis, develop a detailed code review plan, identifying the focus of the review. Experienced auditors perform line-by-line reviews of key components and sensitive functionalities in the code.
- If any issues arise during the audit process, communicate with the code owner in a timely manner. The code owners should actively cooperate (this may include providing the latest stable source code, relevant deployment scripts or methods, transaction signature scripts, exchange docking schemes, etc.);
- Necessary information during the audit process will be well documented in a timely manner for both the audit team and the code owner.

2 Summary

This report has been commissioned by Liquidium with the objective of identifying any potential issues and vulnerabilities within the source code of the Liquidium repository, as well as in the repository dependencies that are not part of an officially recognized library. In this audit, we have employed the following techniques to identify potential vulnerabilities and security issues:

(1) Dependency Check

A comprehensive analysis of the software's dependency libraries was conducted using the dependency check tool.

(2) Automated Static Code Analysis

The code quality was examined using a code scanner.

(3) Fuzz Testing

Based on the fuzz tool and by writing harnesses.

(4) Manual Code Review

Manually reading and analyzing code to uncover vulnerabilities and enhance overall quality.

During the audit, we identified 19 issues of varying severity, listed below.

ID	Title	Severity	Status
LIB-1	ICP Canister Cycles Exhaustion Denial of Service	Medium	Fixed
LIB-2	Stake/Unstake/Withdraw Functions Contain Redundant Base64 Conversion	Informational	Acknowledged

MOD-1	Exchange Rate Manipulation	Medium	Fixed
STA-1	Stake-PSBT Input UTXO Address Validation Missing Causing Asset Loss	Critical	Fixed
STA-2	`validate_values` Function Integer Overflow	Medium	Fixed
STA-3	Bitcoin Rune Protocol `Amount=0` Handling Vulnerability	Medium	Fixed
TPA-1	Integer underflow in rune amount calculation leading to manipulated TxRecord amounts	Critical	Fixed
TPA-2	TRANSACTION_RECORDS Processing Queue Blocking Denial of Service	Major	Fixed
TPA-3	Hardcoded Confirmation Parameters May Lead to Transaction Omission	Minor	Fixed
TPA-4	Unexpected confirmation block calculation in `handle_confirmed_tx`	Minor	Fixed
UHE-1	Incorrect rune_amount field value in `store_unstake_record`	Medium	Fixed
UNS-1	Unstake-PSBT Input UTXO Address Validation Missing Causing Asset Loss	Critical	Fixed

UNS-2	Bitcoin Rune Protocol Large Amount Handling Vulnerability	Critical	Fixed
WHE-1	Incomplete input validation in `extract_withdraw_data`	Critical	Fixed
WIT-1	Missing `rune_amount` validation in withdraw validation	Minor	Fixed
LIB1-1	Bitcoin Rune Protocol Broadcast Output Handling Vulnerability	Medium	Fixed
MOD1-1	Centralization risk	Medium	Acknowledged
STA1-1	PSBT validation only focuses on runes but signs all unsigned inputs	Minor	Fixed
WIT1-1	Lack of `user_address` amount Validation Leading to Fund Theft in Withdraw	Critical	Fixed

3 Participant Process

Here are the relevant actors with their respective abilities within the Liquidium repository : User:

- **stake**: Deposit LIQ runes into the primary pool to receive sLIQ runes based on exchange rate.
- **unstake**: Convert sLIQ runes back to LIQ runes and transfer them to secondary pool.
- **withdraw**: Withdraw LIQ runes from secondary pool after unstaking period.

4 Findings

LIB-1 ICP Canister Cycles Exhaustion Denial of Service

Severity: Medium

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/lib.rs#1

Descriptions:

1. ICP Cycles Consumption Mechanism

In the ICP network, Cycles consumed during Canister execution of computations and network requests are borne by the Canister itself, not the caller. This means:

- Users calling contract functions do not need to pay any fees
- All computation, storage, and network request costs are borne by the Canister
- Once the Canister's Cycles balance is exhausted, the service will completely stop

2. High-Cost Operation Identification

Through code analysis, the following high Cycles consumption operations were identified:

HTTP External Requests (Highest Risk):

- `src/oracle/http.rs:35` - `http_request()` calls external APIs
- `src/oracle/mempool.rs:163-177` - HTTP requests to get transaction information
- `src/oracle/mempool.rs:187-201` - HTTP requests to get transaction hex data

Cross-Canister Calls:

- `src/oracle/omnity.rs:38-41` - Calls Omnity Canister to get Rune balance information
- `src/oracle/bitcoin.rs:26-32` - Calls Bitcoin Canister to get block headers

- `src/oracle/bitcoin.rs:43-51` - Calls Bitcoin Canister to get UTXO information

Complex Computation Operations:

- PSBT signing and verification processes
- Rune data parsing and validation
- Cryptographic computations (address generation, fingerprint calculation, etc.)

Attack Scenarios

Attackers continuously calls `stake()` , `unstake()` , `withdraw()` , or `initialize_pool_addresses_range()` functions

Suggestion:

1. Implement Call Rate Limiting
2. Implement User Payment Mechanism
3. Implement Input lentgh check

```
fn validate_psbt_format(psbt_base64: &str) -> Result<(), String> {
    // Length check
    if psbt_base64.len() > 100_000 { // 100KB limit
        return Err("PSBT data too large".to_string());
    }
    Ok(())
}
```

Resolution:

The client implements input lentgh check.

LIB-2 Stake/Unstake/Withdraw Functions Contain Redundant Base64 Conversion

Severity: Informational

Discovery Methods: Manual Review

Status: Acknowledged

Code Location:

src/lib.rs#1

Descriptions:

In the contract's `stake`, `unstake`, and `withdraw` functions, there exist unnecessary Base64 encoding/decoding conversion operations. These functions receive Base64 string parameters, decode them to byte arrays, and then re-encode them to Base64 strings internally, causing more gas usage.

Redundant Conversion in Stake Function:

```
// src/core/staking.rs:37-39
pub async fn stake(input_psbt_base64: String) -> String {
    // First time: Base64 -> byte array
    let validation_result = validate_stake(&input_psbt_base64).await;

    // Inside validate_stake
    // src/validation/stake.rs:91-93
    let psbt_bytes = BASE64_STANDARD.decode(psbt_base64).map_err(...)?;

    // Second time: byte array -> Base64 (for certain validation functions)
    // src/validation/stake.rs:207-209
    let (rune_data, _debug_output) =
        ordinals_runes::parse_psbt_runes_with_debug_legacy(psbt_base64)...
}
```

Redundant Conversion in Unstake Function:

```
// src/core/unstaking.rs:47-50
pub async fn unstake(input_psbt_base64: String) -> String {
    // First time: Base64 -> byte array
    let decoded = BASE64_STANDARD.decode(&input_psbt_base64)...

    // Second time: byte array -> Base64
    let psbt_base64 = BASE64_STANDARD.encode(psbt_bytes);
}
```

Redundant Conversion in Withdraw Function:

```
// Similar pattern in src/core/withdrawal.rs
// Receive Base64 -> decode -> re-encode -> pass to validation functions
```

Suggestion:

Unify data format processing

MOD-1 Exchange Rate Manipulation

Severity: Medium

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/state/mod.rs#181

Descriptions:

```
pub fn get_stored_exchange_rate() -> Option<f64> {
    match (get_stored_circulating_supply(), get_stored_balance()) {
        (Some(circulating), Some(balance)) => {
            if circulating > 0 {
                Some(balance as f64 / circulating as f64)
            } else {
                Some(1.0)
            }
        }
        _ => None,
    }
}
```

The stored_exchange_rate validation mechanism is vulnerable to exchange rate manipulation attacks, particularly when the pool has low liquidity. Malicious actors can artificially inflate the stored exchange rate at relatively low cost.

- Initial Staking: Attacker stakes 1 LIQ receives 1 SLIQ Initial exchange rate: 1.0
- Rate Manipulation: Attacker stakes 10000 LIQ receives 1 SLIQ Manipulated exchange rate: 10001/2 = 5000.5
- Validation Blocking: Subsequent staking transactions must now meet stake_exchange_ratio >= 5000.5

Suggestion:

When pool balance is below a certain threshold, restrict exchange rate to prevent manipulation

Resolution:

This issue has been fixed. The client has adopted our suggestions.

STA-1 Stake-PSBT Input UTXO Address Validation Missing Causing Asset Loss

Severity: Critical

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/validation/stake.rs#329

Descriptions:

The contract's `stake` interface only checks the amount of Rune tokens in input UTXOs when validating PSBT, but fails to verify the source addresses of input UTXOs. Malicious attackers can construct malicious PSBTs by setting all input UTXOs to come from the Primary Pool address and outputs to the attacker's address, thereby obtaining sLIQ tokens without consuming their own LIQ tokens, causing protocol asset loss.

According to the `doc/PSBT_Stake.md` documentation, a normal Stake transaction should contain:

Expected Input Structure:

```
0. Canister UTXO(s) containing sLIQ runes (amount to be transferred to user)
1. User UTXO(s) containing LIQ runes (amount being staked)
2. User UTXO with BTC for transaction fees (separate from rune UTXOs)
```

Expected Output Structure:

- User receives sLIQ tokens
- Pool address receives user's LIQ tokens
- Various change outputs

`validate_values` **Function Validation Logic:**

```rust
// src/validation/stake.rs:329-389
async fn validate_values(
    psbt: &bitcoin::psbt::PartiallySignedTransaction,
    details: &mut ValidationDetails,
    is_valid: &mut bool,
) {
    // Get all input UTXOs
    let utxos = psbt
        .unsigned_tx
        .input
        .iter()
        .map(|item| item.previous_output.to_string())
        .collect::<Vec<String>>();

    // Only validate LIQ token amount
    let (rune_amount, _) = ord_client
        .get_rune_sent_amount(&utxos, &rune_id)
        .await
        .expect("Could not fetch utxo info");

    // Validate if amount is sufficient
    if rune_amount < details.rune_totals.pool_liq_value {
        details.validation_result = format!("Insufficient stake rune amounts");
        *is_valid = false;
    }
    // ... other amount validation logic
}
```

Key Defects:

1. **Only validates amounts, not sources**: Function only checks if input UTXOs have sufficient LIQ/sLIQ tokens
2. **Missing address ownership check**: Doesn't verify if input UTXOs actually belong to users or pool addresses

Attack

1. Attacker queries Primary Pool address UTXOs, finds UTXOs containing LIQ tokens

2. Constructs malicious PSBT with all inputs set to pool address UTXOs
3. Sets outputs so attacker's address receives sLIQ tokens
4. Since validation function only checks amounts not sources, malicious PSBT passes validation

Specific Attack PSBT Structure:

Malicious Inputs:

- Input 0: Pool_UTXO_1 (contains large amount of sLIQ)
- Input 1: Pool_UTXO_2 (contains large amount of LIQ)
- Input 2: Attacker_BTC_UTXO (for transaction fees)

Malicious Outputs:

- Output 0: OP_RETURN (Runestone)
- Output 1: Pool address (small sLIQ change, creates legitimate appearance)
- Output 2: Pool address (LIQ tokens, creates "user staking to pool" appearance)
- Output 3: Attacker address (large amount of sLIQ tokens)
- Output 4: Attacker address (BTC change)

Suggestion:

Add input UTXO address source validation

Resolution:

This issue has been fixed. The canister only signs utxos that provide "sLIQ" Runes.

STA-2 `validate_values` Function Integer Overflow

Severity: Medium

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/validation/stake.rs#362;

src/validation/unstake.rs#378

Descriptions:

In the contract's `validate_values` function, there exists an integer overflow vulnerability. Attackers can exploit the overflow characteristics of u128 type subtraction operations by setting malicious large values in PSBT Runestone edicts, thereby bypassing validation logic and passing validation checks that should have failed.

The vulnerability exists in the same pattern code in two files:

src/validation/unstake.rs line 362:

```
let expected_liq_change = rune_amount - details.rune_totals.secondary_pool_liq_value;
if expected_liq_change != details.rune_totals.pool_liq_value {
    details.validation_result = format!("Invalid LIQ change");
    *is_valid = false;
}
```

src/validation/stake.rs line 378:

```
let expected_sliq_change = rune_amount - details.rune_totals.user_sliq_value;
if expected_sliq_change != details.rune_totals.pool_sliq_value {
    details.validation_result = format!("Invalid sLIQ change");
    *is_valid = false;
}
```

Trusted Data Source:

- `rune_amount` : Obtained from on-chain UTXO queries via `ord_client.get_rune_sent_amount()` , cannot be forged by users

Untrusted Data Source:

- `details.rune_totals.*` : These values are calculated by parsing Runestone edicts in PSBT, completely controllable by users

Data Types:

- All related variables are `u128` type, which wraps around on overflow

Attack Scenario Example (simplified with u8 to illustrate principle): Assume all data types are u8, attacker constructs the following values:

- `rune_amount = 5` (from real UTXO, cannot be forged)
- `details.rune_totals.user_sliq_value = 255` (forged via edict)
- `details.rune_totals.pool_sliq_value = 6` (forged via edict)

Integer Overflow Calculation:

```
let expected_sliq_change = rune_amount - details.rune_totals.user_sliq_value;
// expected_sliq_change = 5 - 255 = 6 (due to u8 overflow: 5 - 255 = 5 + (256 - 255) = 6)

if expected_sliq_change != details.rune_totals.pool_sliq_value {
   // 6 != 6 is false, validation passes!
}
```

Real Attack Scenario (u128)

For u128 type, attackers can:

1. Set an extremely large `user_sliq_value` (close to u128::MAX)
2. When `rune_amount` is small, subtraction will overflow
3. Through careful calculation, make the overflowed result equal to `pool_sliq_value`
4. Thus bypass validation logic that should detect mismatches

Overflow Calculation Formula:

```
If rune_amount < user_sliq_value:
expected_change = rune_amount + (u128::MAX + 1 - user_sliq_value)
```

Suggestion:

Add value range checks

Resolution:

This issue has been fixed. The client has adopted our suggestions.

STA-3 Bitcoin Rune Protocol `Amount=0` Handling Vulnerability

Severity: Medium

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/validation/stake.rs#240-268;

src/validation/unstake.rs#229-257

Descriptions:

In the Bitcoin Rune protocol, when an edict's `amount` field is 0, it indicates transferring all remaining amounts of that rune type. This behavior is confirmed in the official ordinals library implementation:

```
// From ordinals library:
https://github.com/ordinals/ord/blob/master/src/index/updater/rune_updater.rs#L114
let amount = if amount == 0 {
    *balance  // Transfer all remaining balance
} else {
    amount.min(*balance)  // Transfer specified amount
};
```

However, in the following two critical validation functions, the code directly treats `amount=0` as a transfer amount of 0:

1. `validate_rune_exchange_ratio` function in `src/validation/stake.rs`

```
        if let Some(address) = &edict.address {
            if address == &pool_address {
                if edict.id == liq_rune_id {
                    rune_totals.pool_liq_value += edict.amount;  // Directly adds edict.amount
```

```
            } else if edict.id == sliq_rune_id {
                rune_totals.pool_sliq_value += edict.amount;  // Directly adds edict.amount
            }
        } else {
            // ...
            if edict.id == liq_rune_id {
                rune_totals.user_liq_value += edict.amount;  // Directly adds edict.amount
            } else if edict.id == sliq_rune_id {
                rune_totals.user_sliq_value += edict.amount;  // Directly adds edict.amount
            }
        }
    }
```

2. `validate_rune_exchange_ratio` function in `src/validation/unstake.rs`

Suggestion:

Just return error when `edict.amount=0`

Resolution:

This issue has been fixed. The client has adopted our suggestions.

TPA-1 Integer underflow in rune amount calculation leading to manipulated TxRecord amounts

Severity: Critical

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/core/transaction_parser.rs#237

Descriptions:

The `is_receving_runes` function has a integer underflow vulnerability that can lead to manipulated transaction amounts:

```
let mut residual_rune_amount = rune_amount;
let mut total_recevied = 0;
for edict in runestone_output.edicts {
    if edict.id.to_string() == *rune_id {
        let addr = edict.address.ok_or("missing edict address")?;
        if addr == *receiver {
            // We need to break if there are no more runes available
            if residual_rune_amount < edict.amount {
                total_recevied += residual_rune_amount;
                residual_rune_amount = 0;
                break;
            } else {
                // We have enough runes to keep going
                total_recevied += edict.amount
            }
        }
        residual_rune_amount -= edict.amount;
    }
}
```

According to the Ordinals rune indexer [implementation], when edict.amount exceeds the actual available balance, the indexer uses the minimum of the two values.

When `edict.amount > residual_rune_amount`, the subtraction causes underflow, wrapping `residual_rune_amount` to a very large positive number (close to u128::MAX).

Due to incorrect reward distributions and accounting

Suggestion:

Implement proper overflow and underflow protection and balance validation.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

TPA-2 TRANSACTION_RECORDS Processing Queue Blocking Denial of Service

Severity: Major

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/core/transaction_parser.rs#270

Descriptions:

The Liquidium staking contract contains a TRANSACTION_RECORDS processing queue blocking vulnerability. Malicious attackers can construct large numbers of invalid transaction records to occupy the front of the processing queue, preventing normal transaction records from being processed, causing exchange rate update mechanism failure and reward transaction processing stagnation, resulting in indirect denial of service attacks.

1. Processing Mechanism Design Flaws

Through code review, the following design issues were found in the `process_transaction_records` task:

Fixed Processing Quantity Limit:

```
// src/core/transaction_parser.rs:274-280
TRANSACTION_RECORDS.with_borrow(|transactions| {
    let mut iterator = transactions.iter().take(100);  // Only processes 100 records each time
    while let Some(tx) = iterator.next() {
        let tx_status = self.mempool_client.get_transaction_status(tx.0);
        tx_requests.push(tx_status);
    }
});
```

Invalid Records Not Cleaned:

```
// src/core/transaction_parser.rs:296-358
fn handle_confirmed_tx(&self, tx_status: TxStatus, current_block: u64) {
    match tx_status {
        TxStatus::Confirmed { block_height, txid } => {
            // Only confirmed transactions are deleted
            delete_transaction(&txid);
        }
        _ => {} // TxStatus::NotFound and TxStatus::Unconfirmed do nothing
    }
}
```

2. Attack

Queue Pollution Attack:

1. Attacker submits large numbers of PSBTs through `stake()` , `unstake()` functions
2. Each PSBT generates a `TxRecord` and inserts into `TRANSACTION_RECORDS` after validation
3. Attacker intentionally doesn't broadcast these transactions to Bitcoin network
4. These transactions return `TxStatus::NotFound` status in mempool

Processing Queue Blocking:

1. `process_transaction_records` executes every 300 seconds
2. Only processes the first 100 records each time
3. Since `TxStatus::NotFound` records are not deleted, they permanently occupy the front of the queue
4. Normal transaction records are pushed to the back of the queue and cannot be processed

Suggestion:

It's recommended to implement a timeout cleanup mechanism.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

TPA-3 Hardcoded Confirmation Parameters May Lead to Transaction Omission

Severity: Minor

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/core/transaction_parser.rs#76

Descriptions:

```
let confirmed_transactions = self
    .bitcoin_client
    .get_confirmed_transactions(&address, 4, 7)
    .await
    .map_err(|e| format!("Failed to fetch pool transaction {}", e.to_string()))?;
```

The `scan_for_new_reward_transactions` function uses hardcoded confirmation parameters (4, 7) which may cause transactions with confirmation counts exceeding 7 may be missed when Bitcoin client service is unstable.

Suggestion:

Track the last scanned block height to ensure no transactions are missed.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

TPA-4 Unexpected confirmation block calculation in `handle_confirmed_tx`

Severity: Minor

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/core/transaction_parser.rs#300

Descriptions:

```
if current_block - block_height > CONFIRMATIONS{
}
```

With CONFIRMATIONS = 6, this condition requires current_block - block_height > 6, which means the transaction needs 7 confirmations instead of the intended 6.

Suggestion:

Change the condition to use `>=` instead of `>`.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

UHE-1 Incorrect rune_amount field value in `store_unstake_record`

Severity: Medium

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/core/unstake_helper.rs#31

Descriptions:

```
/// Stores unstake data from validation
pub fn store_unstake_data_from_validation(
    validation_details: &ValidationDetails,
) -> Result<(String, String, u128), UnstakeHelperError> {
    // Validate user address
    let user_address = validation_details
        .user_address
        .as_ref()
        .ok_or(UnstakeHelperError::NoUserAddress)?
        .clone();

    // Validate rune amount
    let rune_amount = validation_details.rune_totals.pool_sliq_value;
    if rune_amount == 0 {
        return Err(UnstakeHelperError::NoSLiqToPool);
    }

    // Validate UTXO
    let utxo = validation_details
        .secondary_pool_utxo
        .as_ref()
        .unwrap_or(&validation_details.rune_totals.sliq_id)
        .clone();
```

```
    if utxo.is_empty() {
        return Err(UnstakeHelperError::NoUtxo);
    }

    // Store the unstake record
    state::store_unstake_record(user_address.clone(), utxo.clone(), rune_amount);

    // Return the data
    Ok((user_address, utxo, rune_amount))
}
```

In the `store_unstake_data_from_validation` function, when calling store_unstake_record to store unstake records, the rune_amount parameter is using an incorrect value.

According to the `UnstakeRecord` struct definition, the `rune_amount` field should represent "Amount of runes transferred from primary to secondary pool".

However, the current code uses `pool_sliq_value` which represents the amount of sLIQ sent to the primary pool, not the amount transferred to the secondary pool.

Suggestion:

Update the field assignment, ensures the UnstakeRecord accurately reflects the runes transferred to the secondary pool.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

UNS-1 Unstake-PSBT Input UTXO Address Validation Missing Causing Asset Loss

Severity: Critical

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/validation/unstake.rs#319

Descriptions:

The contract's `unstake` interface only checks the amount of Rune tokens in input UTXOs when validating PSBT, but fails to verify the source addresses of input UTXOs. Malicious attackers can construct malicious PSBTs by setting all input UTXOs to come from the Primary Pool address and outputs to the attacker's address, thereby obtaining LIQ tokens without consuming their own sLIQ tokens, causing protocol asset loss.

According to the `doc/PSBT_Unstake.md` documentation, a normal Unstake transaction should contain:

Expected Input Structure:

```
0. Canister UTXO(s) containing LIQ runes (amount to be returned to user)
1. User UTXO(s) containing sLIQ runes (amount being unstaked)
2. User UTXO with BTC for transaction fees (separate from rune UTXOs)
```

Expected Output Structure:

- User receives LIQ tokens (through Secondary Pool)
- Primary Pool receives user's sLIQ tokens
- Various change outputs

`validate_values` **Function Validation Logic Defects:**

```rust
// src/validation/unstake.rs:319-367
async fn validate_values(
    psbt: &bitcoin::psbt::PartiallySignedTransaction,
    details: &mut ValidationDetails,
    is_valid: &mut bool,
) {
    // Get all input UTXOs but don't verify address sources
    let utxos = psbt
        .unsigned_tx
        .input
        .iter()
        .map(|item| item.previous_output.to_string())
        .collect::<Vec<String>>();

    // Only validate sLIQ token amount
    let (rune_amount, _) = ord_client
        .get_rune_sent_amount(&utxos, &rune_id)
        .await
        .expect("Could not fetch utxo info");

    // Validate if amount is sufficient, but not the source
    if rune_amount < details.rune_totals.pool_sliq_value {
        details.validation_result = format!("Insufficient stake rune amounts");
        *is_valid = false;
    }
    // ... other amount validation logic
}
```

Key Defects:

1. **Only validates amounts, not sources**: Function only checks if input UTXOs have sufficient sLIQ/LIQ tokens
2. **Missing address ownership check**: Doesn't verify if input UTXOs actually belong to users rather than pool addresses

Attack:

1. Attacker queries Primary Pool address UTXOs, finds UTXOs containing LIQ and sLIQ tokens
2. Constructs malicious PSBT with all inputs set to pool address UTXOs
3. Sets outputs so Secondary Pool address sends LIQ tokens to attacker's address
4. Since validation function only checks amounts not sources, malicious PSBT passes validation

Specific Attack PSBT Structure:

Malicious Inputs:

- Input 0: Pool_UTXO_1 (contains large amount of LIQ)
- Input 1: Pool_UTXO_2 (contains large amount of sLIQ)
- Input 2: Attacker_BTC_UTXO (for transaction fees)

Malicious Outputs:

- Output 0: OP_RETURN (Runestone)
- Output 1: Primary Pool address (small LIQ change)
- Output 2: Primary Pool address (sLIQ tokens, creates "user unstaking" appearance)
- Output 3: Attacker address (sLIQ change)
- Output 4: Secondary Pool address (large amount of LIQ tokens to attacker)
- Output 5: Attacker address (BTC change)

Suggestion:

Add input UTXO address source validation

Resolution:

This issue has been fixed. The canister only signs utxos that provide "LIQ" Runes.

UNS-2 Bitcoin Rune Protocol Large Amount Handling Vulnerability

Severity: Critical

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/validation/unstake.rs#229-257;

src/validation/stake.rs#240-268

Descriptions:

In the Bitcoin Rune protocol, when an edict's `amount` field is greater than the available rune balance in the input UTXOs, the actual transfer amount should be the available balance, not the amount specified in the edict. This behavior is confirmed in the official ordinals library implementation:

```
// From ordinals library:
https://github.com/ordinals/ord/blob/master/src/index/updater/rune_updater.rs#L117
let amount = if amount == 0 {
    *balance  // Transfer all remaining balance
} else {
    amount.min(*balance)  // Transfer specified amount, but capped by available balance
};
```

However, in the following two critical validation functions, the code directly uses `edict.amount` for accumulation without considering the actual transfer amount limitation:

1. `validate_rune_exchange_ratio` function in `src/validation/stake.rs`

```
        if let Some(address) = &edict.address {
            if address == &pool_address {
                if edict.id == liq_rune_id {
                    rune_totals.pool_liq_value += edict.amount;  // Directly adds edict.amount
without capping
                } else if edict.id == sliq_rune_id {
                    rune_totals.pool_sliq_value += edict.amount;  // Directly adds edict.amount
without capping
                }
            } else {
                if user_address.is_none() {
                    user_address = Some(address.clone());
                }
                if edict.id == liq_rune_id {
                    rune_totals.user_liq_value += edict.amount;  // Directly adds edict.amount
without capping
                } else if edict.id == sliq_rune_id {
                    rune_totals.user_sliq_value += edict.amount;  // Directly adds edict.amount
without capping
                }
            }
        }
```

2. `validate_rune_exchange_ratio` function in `src/validation/unstake.rs`

Suggestion:

Implement proper overflow and underflow protection and balance validation.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

WHE-1 Incomplete input validation in `extract_withdraw_data`

Severity: Critical

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/core/withdraw_helper.rs#21

Descriptions:

The `extract_withdraw_data` function has incomplete input validation.

The PSBT may contain multiple input UTXOs, but the function only validates one specific input.

But when `psbt::sign is called` later, it will sign all unsigned inputs regardless of validation.

If the PSBT contains additional input UTXOs that belong to other users' unstake records, the canister will sign them all, potentially allowing unauthorized withdrawals from the secondary pool and causing fund losses.

Suggestion:

Implement comprehensive input validation.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

WIT-1 Missing `rune_amount` validation in withdraw validation

Severity: Minor

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/core/withdrawal.rs#43

Descriptions:

```
fn validate_unstake_record(
    user_address: &str,
    utxo: &str,
    _rune_amount: u128,
) -> Result<(), WithdrawValidationError>
```

In the `validate_unstake_record` function accepts a _rune_amount parameter but never validates it against the stored record.

Additionally, the `extract_withdraw_data` function always returns 0 for the rune amount

Suggestion:

Add validation to ensure the `rune_amount` matches the stored record.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

LIB1-1 Bitcoin Rune Protocol Broadcast Output Handling Vulnerability

Severity: Medium

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/libs/ordinals_runes/src/lib.rs#229

Descriptions:

In the Bitcoin Rune protocol, when an edict's `output` field equals the number of transaction outputs (`output == len(tx.outputs)`), runes are broadcast and distributed to all non-OP_RETURN outputs. This behavior is confirmed in the official ordinals library implementation:

```
// From ordinals library:
https://github.com/ordinals/ord/blob/master/src/index/updater/rune_updater.rs#L81
if output == tx.output.len() {
  // find non-OP_RETURN outputs
  let destinations = tx
    .output
    .iter()
    .enumerate()
    .filter_map(|(output, tx_out)| {
      (!tx_out.script_pubkey.is_op_return()).then_some(output)
    })
    .collect::<Vec<usize>>();

  if !destinations.is_empty() {
    if amount == 0 {
      // if amount is zero, divide balance between eligible outputs
      let amount = *balance / destinations.len() as u128;
      // ... distribute to all eligible outputs
    } else {
```

```
            // if amount is non-zero, distribute amount to eligible outputs
            for output in destinations {
                allocate(balance, amount.min(*balance), output);
            }
        }
    }
}
```

However, this case is not properly handled in the rune parsing library and validation functions:

1. Issue in Rune Parsing Library

```
        // Add address information to edicts
        for edict in &mut edicts {
            if (edict.output as usize) < tx.output.len() {
                let output = &tx.output[edict.output as usize];
                if let Ok(address) =
                    Address::from_script(&output.script_pubkey, Network::Bitcoin)
                {
                    edict.address = Some(address.to_string());
                } else if let Ok(address) =
                    Address::from_script(&output.script_pubkey, Network::Testnet)
                {
                    edict.address = Some(address.to_string());
                }
            }
        }
```

When `edict.output >= tx.output.len()`, `edict.address` remains `None`.

2. Issue in Validation Functions

```
    if let Some(address) = &edict.address {
        if address == &pool_address {
```

```
            if edict.id == liq_rune_id {
                rune_totals.pool_liq_value += edict.amount;
            } else if edict.id == sliq_rune_id {
                rune_totals.pool_sliq_value += edict.amount;
            }
        } else {
            if user_address.is_none() {
                user_address = Some(address.clone());
            }
            if edict.id == liq_rune_id {
                rune_totals.user_liq_value += edict.amount;
            } else if edict.id == sliq_rune_id {
                rune_totals.user_sliq_value += edict.amount;
            }
        }
    }
```

The validation functions only process edicts with address information, completely ignoring broadcast output cases.

Suggestion:

Just return error when `edict.output == len(tx.outputs)`

Resolution:

This issue has been fixed. The client has adopted our suggestions.

MOD1-1 Centralization risk

Severity: Medium

Discovery Methods: Manual Review

Status: Acknowledged

Code Location:

src/oracle/mod.rs#1-5

Descriptions:

The Liquidium staking canister's normal operation depends on multiple some centralized services:

```
pub mod ord_client;
pub mod omnity;
pub mod mempool;
```

Suggestion:

Reduces dependency on single external services and improves resilience.

STA1-1 PSBT validation only focuses on runes but signs all unsigned inputs

Severity: Minor

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/core/staking.rs#117

Descriptions:

In the current PSBT validation implementation, the system only validates and focuses on LIQ and sLIQ runes, but the `psbt::sign` function will sign all unsigned inputs in the PSBT, including non-rune UTXOs.

Suggestion:

Validate all inputs are legitimate for the operation.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

WIT1-1 Lack of `user_address` amount Validation Leading to Fund Theft in Withdraw

Severity: Critical

Discovery Methods: Manual Review

Status: Fixed

Code Location:

src/validation/withdraw.rs#62

Descriptions:

In the withdraw business logic, the `validate_unstake_record` function has flaws in user address identification and validation that could allow malicious users to steal funds from other users.

Vulnerability Analysis

1. User Address Identification Logic

```
    // Find user address in transaction outputs
    for output in psbt.unsigned_tx.output.iter() {
        if let Some(address) =
            bitcoin::Address::from_script(&output.script_pubkey,
crate::state::get_btc_network())
                .ok()
        {
            let addr_str = address.to_string();
            // Skip pool address and OP_RETURN outputs
            if addr_str != secondary_pool_address && !output.script_pubkey.is_op_return() {
                user_address = Some(addr_str);
                break;
            }
```

```
        }
    }
```

This function identifies the first non-secondary_pool_address and non-op_return output address as the user address, but does not verify whether that address actually owns sufficient LIQ runes.

2. Incomplete Validation Logic

```
fn validate_unstake_record(
    user_address: &str,
    utxo: &str,
    _rune_amount: u128,
) -> Result<(), WithdrawValidationError> {
    let records = state::get_user_unstake_records(user_address);

    // Find record with matching UTXO
    let record = records.iter().find(|r| r.utxo == utxo);

    // ... validation logic ...

    if record.user_address != user_address {
        return Err(WithdrawValidationError::UserAddressMismatch(
            record.user_address.clone(),
            user_address.to_string(),
        ));
    }

    if record.utxo != utxo {
        return Err(WithdrawValidationError::UtxoMismatch(
            record.utxo.clone(),
            utxo.to_string(),
        ));
    }
```

The validation function only checks whether the user address and UTXO match the stored records, but does not verify whether the UTXO corresponding to the user address actually contains sufficient LIQ runes.

Attack Scenario

A malicious user B can construct the following PSBT to steal funds from a legitimate user A:

Input:

1. secondary_pool utxo (LIQ runes)
2. user_B utxo (empty or minimal BTC)

Output:

1. op_return (transfer LIQ runes to user_B)
2. user_A utxo (546 sat, empty - used to trick address detection)
3. user_B utxo (receives LIQ runes)
4. user_B utxo (change)

Since the `extract_withdraw_data` function will identify user_A's address as the user address, and the validation function only checks address matching without verifying rune amounts, the attacker can successfully steal funds.

Suggestion:

Check `user_address` runes amount.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

Appendix 1

Issue Level

- **Informational** issues are often recommendations to improve the style of the code or to optimize code that does not affect the overall functionality.
- **Minor** issues are general suggestions relevant to best practices and readability. They don't post any direct risk. Developers are encouraged to fix them.
- **Medium** issues are non-exploitable problems and not security vulnerabilities. They should be fixed unless there is a specific reason not to.
- **Major** issues are security vulnerabilities. They put a portion of users' sensitive information or assets at risk, and often are not directly exploitable. All major issues should be fixed.
- **Critical** issues are directly exploitable security vulnerabilities. They put users' sensitive information or assets at risk. All critical issues should be fixed.

Issue Status

- **Fixed:** The issue has been resolved.
- **Partially Fixed:** The issue has been partially resolved.
- **Acknowledged:** The issue has been acknowledged by the code owner, and the code owner confirms it's as designed, and decides to keep it.

Appendix 2

Disclaimer

This report is based on the scope of materials and documents provided, with a limited review at the time provided. Results may not be complete and do not include all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your own risk. A report does not imply an endorsement of any particular project or team, nor does it guarantee its security. These reports should not be relied upon in any way by any third party, including for the purpose of making any decision to buy or sell products, services, or any other assets. TO THE FULLEST EXTENT PERMITTED BY LAW, WE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, IN CONNECTION WITH THIS REPORT, ITS CONTENT, RELATED SERVICES AND PRODUCTS, AND YOUR USE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NOT INFRINGEMENT.