Crato Liquidity Protocol Finance
Security Audit Report

Prepared by VelintSec
Version 2.0

Lead Auditors
Vincent
MaxMa

February 24th ,2025

Contents

1 About VelintSec

VelintSec is a full-service security service company based on the blockchain and Web3 ecosystem, focusing on the ICP ecosystem, focusing on providing security audit, protection and consulting services for smart contracts, blockchain projects and centralized systems.Our mission is to establish and maintain the highest security standards in the blockchain industry through rigorous auditing, continuous innovation, and dedicated support.. Learn more about us at velintsec.com.

2 Disclaimer

The VelintSec team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the The Internet Computer (ICP) implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Crato Liquidity Protocol (CLP) is a decentralized financial protocol built on the Internet Computer Protocol (ICP) ecosystem, providing users with low-cost, interest-free lending services to enhance the liquidity and value of crypto assets..

DEPOSIT

Users can efficiently deposit ckETH or ckBTC into the CLP platform and instantly receive the corresponding CUSD stablecoin. The deposit process is straightforward and efficient, eliminating complex procedures, with real-time calculations of the generated CUSD quantity.

LOAN

After depositing a certain amount of ckETH or ckBTC, users can borrow the equivalent value in CUSD. CLP follows a zero-interest lending policy, requiring users to pay a one-time fee, providing a flexible and stress-free borrowing service.

LIQUIDATION

CLP features a liquidity pool for clearing pledged assets, ensuring system stability. Users can earn rewards by providing liquidation services, gaining additional CLPT tokens from the liquidity pool.

CLPT TOKEN

Users participating in liquidation by depositing CUSD into the liquidity pool receive CLPT token rewards. CLPT also serves as a governance token, allowing holders to vote and propose governance changes.

GOVERNANCE

Users holding CLPT tokens are eligible to participate in CLP governance. They can vote on issues related to platform upgrades, parameter adjustments, and actively engage in shaping the platform's future.

COMMUNITY

CLP encourages user involvement in community building. Users can actively contribute to the platform's community prosperity by proposing innovative ideas, participating in discussions, and more.

Tokenomics

CUSD Stablecoin

- Issuance Mechanism: CUSD is a stablecoin issued based on users pledging ckETH or ckBTC, with its supply directly linked to the value of the collateral pledged by users. Users can obtain CUSD by depositing a certain quantity of ckETH or ckBTC, following the collateralization ratio.

- Zero-Interest Lending: CLP adopts a zero-interest lending policy, enabling users to borrow CUSD without incurring high-interest costs. Users only need to pay a one-time fee, making borrowing services more accessible and economical.

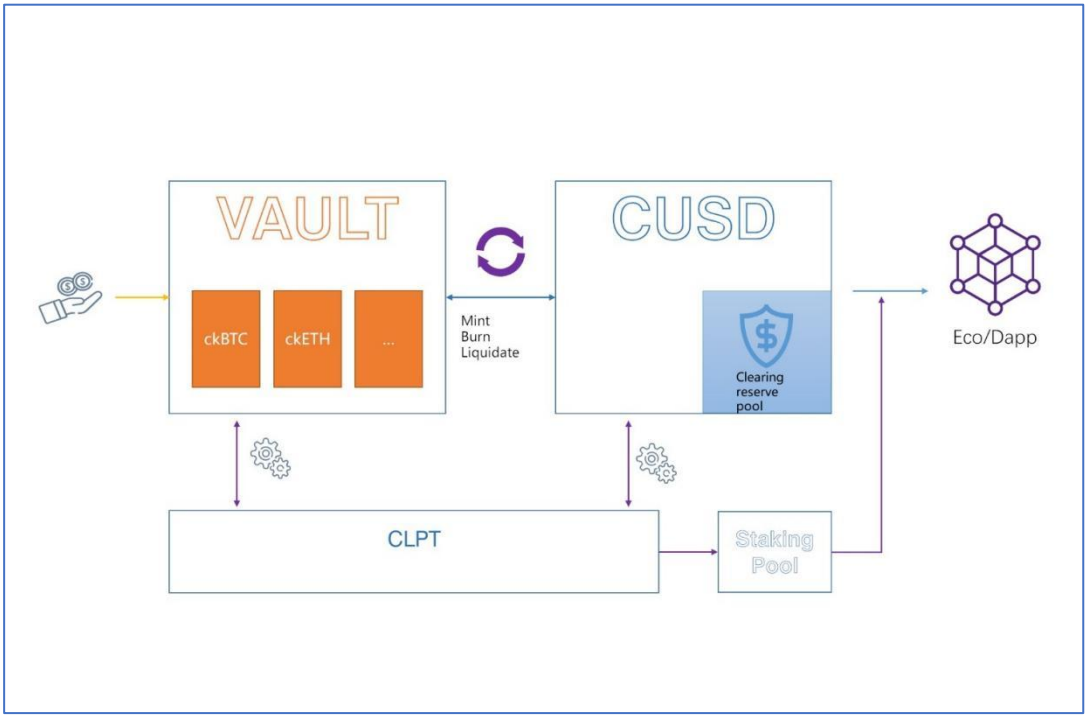
CLPT Token

- Total Supply: The total supply of CLPT is 1 billion tokens, with 20% allocated to sponsors and investment institutions, 10% to the product technical team, and 10% for marketing promotions. The remaining 60% is gradually released to the community through mining rewards.

- Mining Rewards: Users engaging in activities such as deposits, loans, liquidation, and providing liquidity to the liquidity pool have the opportunity to earn CLPT mining rewards. Mining serves as a mechanism to incentivize users to actively contribute to community development and share in the platform's success.

- Governance Rights: CLPT functions as the governance token of the CLP platform. Holders have the right to participate in governance decisions, including voting for or against proposals, participating in parameter adjustments, allowing for deeper user involvement in the platform's future.

- Community Incentives: A portion of CLPT tokens is allocated for incentivizing community development. Users contributing innovative ideas and actively participating in discussions receive rewards, fostering a healthy and engaged community.



5 Audit Scope

The following contracts were included in the scope for this audit:

- Audit Scope: All source code
- Main Modules:
  - vault
  - staking_pool
  - reserve_pool
  - icrc1

6 Executive Summary

Over the course of 12 days, the VelintSec team conducted an audit on the CLPFinance smart contracts provided by CLP. In this period, a total of 12 issues were found.

The findings consist of 0 Critical, 0 High, 4 Medium, 2 Low severity and 6 Business Logic issues with the remainder being informational.

Summary

Project Name	clp-rust-main
Repository	clp-rust
Commit	74697994e92a75b46a6e204dba21680876aabfdd. . .
Audit Timeline	January 2nd - 14th, 2025
Methods	Manual Review

Issues Found

Critical Risk	**0**
High Risk	**0**
Medium Risk	**4**
Low Risk	**2**
Business Logic Issues	**6**
Total Issues	**12**

Summary of Findings

[Medium]--State Synchronization Issue, liquidate() in src/reserve_pool/src/liquidate.rs	Resolved
[Medium]--Insufficient Access Control, withdrawFee() in src/vault/src/operations.rs	Resolved
[Medium]--Funds Locking Risk, unstake() in src/staking_pool/src/clpt_staking.rs	Resolved
[Medium]--Timestamp Dependency Issue, collateral_inspection() in src/staking_pool/src/time_task.rs	Resolved
[Low]--Repeated State Read, withdraw() in src/vault/src/withdraw.rs	Resolved
[Low]--Asset Allocation Inequality, liquidate() in src/reserve_pool/src/liquidate.rs	Resolved
[Logic]--Oracle Risk Issue	Acknowledged
[Logic]--Insufficient CUSD in the Liquidation Pool during Liquidation	Acknowledged
[Logic]--Race Condition in withdraw Function, withdraw() in src/vault/src/withdraw.rs	Resolved
[Logic]--Price Manipulation Vulnerability, filter_liquidate_user() in src/vault/src/time_task.rs	Resolved
[Logic]--Liquidation Calculation Precision Issue, calculate_liquidate_amount() in src/vault/src/time_task.rs	Resolved
[Logic]--Numerical Overflow Risk, withdraw() in src/staking_pool/src/mining.rs	Resolved

7 Findings

7.1 Medium Risk

7.1.1 State Synchronization Issue

Description: The code contains a state synchronization vulnerability because it reads and updates the state in a non-atomic manner. Specifically, the code first iterates over state.borrow().users to populate update_list, and then updates state.users using update_list. During the gap between these operations, the state could be modified by another transaction, leading to inconsistent or incorrect updates.

Impact: The state synchronization vulnerability can result in:

```
// liquidate() in src/reserve_pool/src/liquidate.rs
STATE.with(|state| {
    if cusd_total > 0 {
        let mut update_list: HashMap<Principal, UserInfo> = HashMap::new();
        for (addr, mut info) in state.borrow().users.iter() {
            // ... state update logic
        }
        let mut state = state.borrow_mut();
        for (addr, info) in update_list {
            state.users.insert(addr, info);
        }
    }
})
```

- Inconsistent State: The state may become corrupted or inconsistent due to concurrent modifications.
- Data Loss: Updates from other transactions may be overwritten or lost.
- Exploitation: Attackers could exploit the vulnerability to manipulate the state for their benefit.

Recommended Mitigation: To fix the state synchronization vulnerability, ensure that state updates are performed atomically. Below are two recommended solutions with realistic code examples:

- Read and update the state within a single borrow operation to ensure atomicity.

```
STATE.with(|state| {
    if cusd_total > 0 {
        let mut state = state.borrow_mut();
        for (addr, mut info) in state.users.iter_mut() {
            // Perform state update logic directly
            // Example: Update user info
            info.balance += 100; // Example update
        }
    }
});
```

- Collect updates in a temporary variable and apply them in a single operation to ensure consistency.

```
STATE.with(|state| {
    if cusd_total > 0 {
        let mut update_list: HashMap<Principal, UserInfo> = HashMap::new();
        {
            let state = state.borrow();
            for (addr, info) in state.users.iter() {
                // Populate update_list based on current state
                let mut updated_info = info.clone();
                updated_info.balance += 100; // Example update
                update_list.insert(addr.clone(), updated_info);
            }
        }

        // Apply updates in a single operation
        let mut state = state.borrow_mut();
        for (addr, info) in update_list {
            state.users.insert(addr, info);
        }
    }
});
```

CLP: Fixed @ commit 9b46822

VelintSec: Resolved

7.1.2 Insufficient Access Control

Description: The code contains insufficient access control because it only performs a simple custodian check (guard = "check_custodian") without verifying additional conditions, such as whether the caller is authorized to withdraw fees or whether the withdraw amount is valid. This could allow unauthorized users or malicious actors to exploit the function and withdraw fees improperly.

```
// withdrawFee() in src/vault/src/operations.rs
#[update(name = "withdrawFee", guard = "check_custodian")]
async fn withdraw_fee(amount: u128) -> Result<()> {
    // Only simple custodian check
    let cusd = STATE.with(|state| {
        let state = state.borrow();
        if amount > state.fee_balance {
            Err(Error::InsufficientBalance)
        } else {
            Ok(state.cusd.unwrap())
        }
    })?;
    // ...
}
```

Impact: The insufficient access control vulnerability can result in:

- Unauthorized Fee Withdraws: Malicious actors could withdraw fees without proper authorization.
- Financial Loss: The protocol could lose funds if fees are withdrawn improperly.
- Protocol Instability: Repeated unauthorized withdraws could undermine trust in the protocol and destabilize its operations.

Recommended Mitigation: To fix the insufficient access control vulnerability, implement stricter access control checks to ensure that only authorized users can withdraw fees and that the withdraw amount is valid. Below are two recommended solutions with realistic code examples:

- Use a role-based access control system to ensure that only authorized users (e.g., custodians or admins) can call the withdraw_fee function.

```rust
#[update(name = "withdrawFee", guard = "check_custodian")]
async fn withdraw_fee(amount: u128) -> Result<()> {
    let caller = api::caller();
    STATE.with(|state| {
        let state = state.borrow();

        // Check if the caller is a custodian
        if !state.custodians.contains(&caller) {
            return Err(Error::Unauthorized);
        }

        // Check if the withdrawal amount is valid
        if amount > state.fee_balance {
            return Err(Error::InsufficientBalance);
        }

        Ok(())
    })?;

    // Perform the withdrawal
    let cusd = STATE.with(|state| state.borrow().cusd.unwrap());
    // ...
}
```

- Require multiple signatures (e.g., from custodians or admins) to approve fee withdraws, adding an extra layer of security.

```rust
#[update(name = "withdrawFee", guard = "check_custodian")]
async fn withdraw_fee(amount: u128, signatures: Vec<Principal>) -> Result<()> {
    let caller = api::caller();
    STATE.with(|state| {
        let state = state.borrow();

        // Check if the caller is a custodian
        if !state.custodians.contains(&caller) {
            return Err(Error::Unauthorized);
        }

        // Check if the required number of signatures is provided
        let required_signatures = 2; // Example: Require 2 signatures
        if signatures.len() < required_signatures {
            return Err(Error::InsufficientSignatures);
        }

        // Verify the signatures
        for signer in signatures {
            if !state.custodians.contains(&signer) {
                return Err(Error::InvalidSignature);
            }
        }

        // Check if the withdrawal amount is valid
        if amount > state.fee_balance {
            return Err(Error::InsufficientBalance);
        }

        Ok(())
    })?;

    // Perform the withdrawal
    let cusd = STATE.with(|state| state.borrow().cusd.unwrap());
    // ...
}
```

CLP: Fixed @ commit [9b46822]
VelintSec: Resolved

7.1.3 Funds Locking Risk

Description: The code contains an asset locking vulnerability because it does not properly enforce the locking period for staked assets. Specifically, the code checks if the current

time (now) is greater than the user's staking period end (user.period_end), but it does not ensure that the user's assets are locked during the staking period. This could allow users to bypass the locking mechanism and withdraw their assets prematurely.

```
// unstake() in src/staking_pool/src/clpt_staking.rs
if user.status == Status::Valid {
        if now > user.period_end {
                Ok((user.balance, user.period))
        } else {
                Err(Error::InStaking)
        }
} else {
        Err(Error::Invalid)
}
```

Impact: The asset locking vulnerability can result in:

- Premature Withdraws: Users could withdraw their staked assets before the locking period ends, undermining the staking mechanism.
- Financial Loss: The protocol could lose funds if users withdraw assets prematurely and fail to fulfill their staking commitments.
- Protocol Instability: Repeated premature withdraws could destabilize the protocol and reduce trust among users.

Recommended Mitigation: To fix the asset locking vulnerability, enforce the locking period more strictly and ensure that users cannot withdraw their assets before the staking period ends. Below are two recommended solutions with realistic code examples:

- Add a state check to ensure that the user's assets are locked during the staking period.

```
if user.status == Status::Valid {
    if now > user.period_end {
        // Ensure the user's assets are unlocked before allowing withdrawal
        if user.is_locked {
            return Err(Error::AssetsLocked);
        }
        Ok((user.balance, user.period))
    } else {
        // Lock the user's assets during the staking period
        user.is_locked = true;
        Err(Error::InStaking)
    }
} else {
    Err(Error::Invalid)
}
```

- Implement a time-based locking mechanism that prevents withdraws until

the staking period ends.

```
if user.status == Status::Valid {
    if now > user.period_end {
        // Allow withdrawal only if the staking period has ended
        Ok((user.balance, user.period))
    } else {
        // Prevent withdrawal during the staking period
        return Err(Error::InStaking);
    }
} else {
    Err(Error::Invalid)
}
```

CLP: Fixed @ commit 9b46822
VelintSec: Resolved

7.1.4 Timestamp Dependency Issue

Description: The code contains a timestamp dependency issue because it relies on the current time (ic_cdk::api::time()) to calculate the start of the day (today). This makes the code vulnerable to manipulation, as the timestamp can be influenced by miners or malicious actors. Additionally, the calculation of today using now % 86400 * SECONDS is incorrect due to operator precedence, which could lead to logical errors.

```
// collateral_inspection() in src/staking_pool/src/time_task.rs
let now = ic_cdk::api::time();
let today = now - now % 86400*SECONDS;
STATE.with(|s| {
    let mut state = s.borrow_mut();
    // ... timestamp-based logic
})
```

Impact: The timestamp dependency issue can result in:

- Manipulation of Timestamps: Malicious actors could manipulate the timestamp to exploit time-based logic, such as claiming rewards or unlocking assets prematurely.
- Logical Errors: Incorrect calculation of today due to operator precedence could lead to unexpected behavior, such as incorrect time-based calculations.
- Financial Loss: Exploitation of timestamp manipulation could result in financial losses for the protocol or its users.

Recommended Mitigation: To fix the timestamp dependency issue, avoid relying on block timestamps for critical logic and use safer alternatives. Below are two recommended solutions with realistic code examples:

- Replace ic_cdk::api::time() with a secure time source, such as a decentralized oracle (e.g., Chainlink) or a trusted external service.

```
// Fetch time from a decentralized oracle
let now = fetch_secure_time().await;
// Corrected calculation with proper parentheses
let today = now - (now % (86400 * SECONDS));

STATE.with(|s| {
    let mut state = s.borrow_mut();
    // ... timestamp-based logic
});
```

- Replace timestamp-based logic with block height-based logic, as block height is less susceptible to manipulation.

```
// Use block height instead of timestamp
let current_block = ic_cdk::api::block_height();
// Calculate blocks per day based on average block time
let blocks_per_day = 86400 * SECONDS / BLOCK_TIME;

STATE.with(|s| {
    let mut state = s.borrow_mut();
    let start_of_day_block = current_block - (current_block % blocks_per_day);
    // ... block height-based logic
});
```

CLP: Fixed @ commit [9b46822]
VelintSec: Resolved

7.2 Low Risk

7.2.1 Repeated State Read

Description: The code contains a repeated state read issue because it reads the state multiple times (STATE.with) in separate operations. This can lead to inconsistent state views, as the state might change between reads. Additionally, repeated state reads can increase gas costs and reduce efficiency.

```
// withdraw() in src/vault/src/withdraw.rs
// First state read
let (need_fee, fee) = STATE.with(|state| {
    let state = state.borrow();
    // ... balance check
})?;

// Second state read
let staking = Staking::new(STATE.with(|state| state.borrow().staking_pool).unwrap());

// Third state read
STATE.with(|state| {
    let mut state = state.borrow_mut();
    // ... state update
})
```

Impact: The repeated state read issue can result in:

- Inconsistent State: The state might change between reads, leading to inconsistent or incorrect behavior.
- Increased Gas Costs: Repeated state reads can increase transaction costs, especially in blockchain environments where gas fees apply.
- Reduced Efficiency: Multiple state reads can slow down the execution of

the function, reducing overall efficiency.

Recommended Mitigation: To fix the repeated state read issue, minimize the number of state reads by consolidating them into a single operation. Below are two recommended solutions with realistic code examples:

- Read the state once and store the necessary data in local variables for later use.

```
STATE.with(|state| {
    let mut state = state.borrow_mut();

    // Perform all state reads in a single operation
    let (need_fee, fee) = {
        // ... balance check logic
        (need_fee, fee)
    };

    let staking_pool = state.staking_pool.clone(); // Read staking pool
    let staking = Staking::new(staking_pool);

    // Perform state updates
    // ... state update logic

    Ok(())
})?;
```

- Create a snapshot of the state at the beginning of the function and use it for all subsequent operations.

```
let state_snapshot = STATE.with(|state| {
    let state = state.borrow();
    (
        state.need_fee, // Example: Read need_fee
        state.fee,      // Example: Read fee
        state.staking_pool.clone(), // Example: Read staking_pool
    )
});

let (need_fee, fee) = {
    // ... balance check logic using state_snapshot
    (state_snapshot.0, state_snapshot.1)
};

let staking = Staking::new(state_snapshot.2); // Use staking_pool from snapshot

STATE.with(|state| {
    let mut state = state.borrow_mut();
    // ... state update logic
})?;
```

CLP: Fixed @ commit 9b46822
VelintSec: Resolved

7.2.2 Asset Allocation Inequality

Description: The code contains a potential business logic issue because it calculates item_cusd_sub and item_asset_add using proportional division (info.balance * transfer_cusd / cusd_total and info.balance * transfer_asset / cusd_total). However, if cusd_total is zero or very small, this could lead to division by zero or significant precision loss, resulting in incorrect calculations. Additionally, the logic does not handle cases where info.balance, transfer_cusd, or transfer_asset might cause overflow during multiplication.

```
// liquidate() in src/reserve_pool/src/liquidate.rs
let item_cusd_sub = info.balance * transfer_cusd / cusd_total;
let item_asset_add = info.balance * transfer_asset / cusd_total;
```

Impact: The business logic issue can result in incorrect calculations, financial losses, or protocol instability due to division by zero, precision loss, or integer overflow, potentially leading to unfair distributions or user dissatisfaction.

Recommended Mitigation: Add checks to prevent division by zero and use safe arithmetic operations to avoid overflow and precision loss.

```
let item_cusd_sub = if cusd_total == 0 {
        0 // Handle division by zero
} else {
        info.balance.checked_mul(transfer_cusd)
                .and_then(|v| v.checked_div(cusd_total))
                .unwrap_or(0) // Handle overflow
};
let item_asset_add = if cusd_total == 0 {
        0 // Handle division by zero
} else {
        info.balance.checked_mul(transfer_asset)
                .and_then(|v| v.checked_div(cusd_total))
                .unwrap_or(0) // Handle overflow
};
```

CLP: Fixed @ commit [9b46822]

VelintSec: Resolved

7.3 Business Logic Issues

7.3.1 Oracle Risk Issue

Description: This risk occurs when price oracles used for liquidation mechanisms fail to reflect real-time market prices during extreme BTC price crashes. Decentralized protocols often rely on external oracles (e.g., Chainlink, Band Protocol) to fetch asset prices, but during a BTC crash, liquidations triggered by outdated oracle prices result in mispriced collateral seizures, harming users or leaving the protocol undercollateralized.

Impact: This oracle risk exposes protocols to unfair liquidations (executed at stale prices during BTC crashes) and insolvency (from undetected undercollateralization), while enabling MEV bots to exploit price gaps for profit, ultimately eroding user trust.

Recommended Mitigation: Protocols must adopt multi-oracle aggregation with volatility-based weighting (prioritizing low-latency feeds during crashes), enforce liquidity-backed price validation to reject unreliable data, implement volatility circuit breakers to halt liquidations during extreme swings, deploy decentralized fallback oracles, and provide real-time transparency on oracle performance to users..

CLP: Our protocol follows industry-standard practices by relying on established oracle solutions (e.g., Chainlink, Band Protocol) for price feeds. Like most DeFi projects, we prioritize oracle reliability and decentralization over real-time market synchronization, as frequent on-chain price updates would incur unsustainable gas costs and latency tradeoffs. While oracle latency during extreme volatility is a known industry-wide limitation, we believe our current setup aligns with mainstream risk management frameworks.

VelintSec: Acknowledged

7.3.2 Insufficient CUSD in the Liquidation Pool during Liquidation

Description: This risk involves insufficient CUSD liquidity in the protocol's liquidation pool during market stress, which may prevent the full execution of undercollateralized position liquidations. In DeFi protocols, liquidation relies on third-party actors (e.g., keepers or automated systems) to repay debt in exchange for discounted collateral. However, if the liquidation pool lacks CUSD reserves to cover the debt of underwater positions, liquidations may stall, leading to accumulated bad debt. This scenario is exacerbated during extreme volatility, where multiple positions fall below the liquidation threshold simultaneously, overwhelming the pool's capacity.

Impact: The primary harm lies in cascading protocol insolvency and market contagion. If the CUSD pool cannot absorb liquidation demand, bad debt accrues, eroding the protocol' s solvency and user trust..

Recommended Mitigation: Establish a dedicated reserve fund, capitalized via protocol revenue or overcollateralization, to backstop the CUSD pool during shortages.

CLP: To address liquidation pool risks, we collaborate with trusted liquidity partners to maintain CUSD reserves during volatility, allocate an emergency fund from protocol reserves, and deploy real-time monitoring systems to trigger automated safeguards (e.g., liquidity activation) if pool thresholds or prices deviate. This approach ensures stability while preserving decentralized governance.

VelintSec: Acknowledged

7.3.3 Race Condition in withdraw Function

Description: The code has a race condition vulnerability because it separates the balance check and the transfer operation into non-atomic steps. During the gap between these steps, the state (e.g., user balances) could be modified by another operation, leading to inconsistencies such as double-spending or insufficient balance errors.

```
// withdraw() in src/vault/src/withdraw.rs
let (need_fee, fee) = STATE.with(|state| {
    let state = state.borrow();
    // Check balance
    match state.balances.get(&identify) {
        Some(info) => {
            if info.balance < amount {
                Err(Error::InsufficientBalance)
            } else {
                Ok((need_fee, fee))
            }
        }
        None => Err(Error::InsufficientBalance),
    }
})?;
// ... other operations in between
// Actual transfer
token.transfer(TransferArgsICRC1 { ... })
```

Impact: The race condition can result in:

- Double-Spending: A user could initiate multiple transfers simultaneously, causing the contract to process transfers even if the balance is insufficient.
- Inconsistent State: The contract's state (e.g., balances) may become corrupted or incorrect.
- Financial Loss: Users may lose funds due to incorrect balance calculations or unauthorized transfers.

Recommended Mitigation: To fix the race condition vulnerability, ensure that the balance check and transfer operation are performed atomically. Below are the two recommended solutions:

- Use a mutex to enforce atomic access to shared state, preventing concurrent modifications, introduce a mutex to ensure that only one operation can modify the shared state at a time. This prevents concurrent operations from interfering with each other.

```rust
use std::sync::Mutex;

struct Canister {
    balances: Mutex<std::collections::BTreeMap<Principal, u64>>, // Shared state protected by a mutex
    fee_balance: u64,
}

impl Canister {
    async fn transfer(&self, identify: Principal, amount: u64) -> Result<(), Error> {
        // Acquire the lock to ensure exclusive access to the shared state
        let mut balances = self.balances.lock().unwrap();

        // Check balance and deduct amount atomically
        let user_balance = balances.get_mut(&identify).ok_or(Error::InsufficientBalance)?;
        if *user_balance < amount {
            return Err(Error::InsufficientBalance);
        }
        *user_balance -= amount;

        // Perform the transfer (external call)
        token.transfer(TransferArgsICRC1 {
            amount: Nat::from(amount - need_fee),
            // ...
        })
        .await
        .map_err(|_| Error::TransferFailed)?;

        // Update fee balance
        self.fee_balance += need_fee - fee;

        Ok(())
    }
}
```

- Ensure state changes are completed before external calls, eliminating the gap where race conditions can occur. Follow the Checks-Effects-Interactions pattern to ensure that all state changes are made before any external calls or interactions. This eliminates the gap between the balance check and the transfer operation.

```
STATE.with(|state| {
    let mut state = state.borrow_mut();

    // Check balance and deduct amount atomically
    let user_info = state.balances.get_mut(&identify).ok_or(Error::InsufficientBalance)?;
    if user_info.balance < amount {
        return Err(Error::InsufficientBalance);
    }
    user_info.balance -= amount;

    // Update fee balance
    state.fee_balance += need_fee - fee;

    // Perform the transfer (external call)
    token.transfer(TransferArgsICRC1 {
        amount: Nat::from(amount - need_fee),
        // ...
    })
    .await
    .map_err(|_| Error::TransferFailed)?;

    Ok(())
})
```

CLP: **Fixed @ commit [9b46822]**

VelintSec: **Resolved**

7.3.4 Price Manipulation Vulnerability

Description: The code contains a price manipulation vulnerability because it directly uses the current price (underlying_info.price) from the state for liquidation calculations without verifying the price's authenticity or freshness. An attacker could manipulate the price (e.g., through a flash loan or oracle attack) to trigger unfair liquidations or exploit the protocol for profit.

```
// filter_liquidate_user() in src/vault/src/time_task.rs
fn filter_liquidate_user(
    underlying_updates: &mut HashMap<Principal, (u128, u128, u128, Vec<Principal>)>,
) -> Principal {
    STATE.with(|state| {
        let state = state.borrow();
        // Directly use current price for liquidation
        let underlying_info = state.underlying.get(&asset).unwrap();
        let liquidate_amount = calculate_liquidate_amount(
            info.balance,
            underlying_info.price,
            underlying_info.liquidate_rate,
            decimals,
        );
        // ...
    })
}
```

Impact: The price manipulation vulnerability can result in:

- Unfair Liquidations: Users may be unfairly liquidated if the price is manipulated to make their positions appear undercollateralized.
- Financial Loss: Attackers can exploit the vulnerability to profit at the expense of other users or the protocol.
- Protocol Instability: Repeated price manipulation attacks can undermine trust in the protocol and destabilize its operations.

Recommended Mitigation: To mitigate the price manipulation vulnerability, implement safeguards to ensure the price used for liquidation is accurate and resistant to manipulation. Below are two recommended solutions with realistic code examples:

- Replace the current price with a Time-Weighted Average Price (TWAP) to smooth out short-term price fluctuations and reduce the impact of manipulation.

```
fn filter_liquidate_user(
    underlying_updates: &mut HashMap<Principal, (u128, u128, u128, Vec<Principal>)>,
) -> Principal {
    STATE.with(|state| {
        let state = state.borrow();
        let underlying_info = state.underlying.get(&asset).unwrap();

        // Use TWAP instead of the current price
        let twap_price = calculate_twap_price(&underlying_info.price_history);
        let liquidate_amount = calculate_liquidate_amount(
            info.balance,
            twap_price, // Use TWAP price
            underlying_info.liquidate_rate,
            decimals,
        );
        // ...
    })
}
// Helper function to calculate TWAP
fn calculate_twap_price(price_history: &[(u64, u128)]) -> u128 {
    let total_weighted_price = price_history.iter().map(|(timestamp, price)| price *
timestamp).sum::<u128>();
    let total_timestamps = price_history.iter().map(|(timestamp, _)| timestamp).sum::<u64>();
    total_weighted_price / total_timestamps
}
```

- Fetch the price from a decentralized oracle (e.g., Chainlink) and validate it against multiple sources to ensure its accuracy and resistance to manipulation.

```
fn filter_liquidate_user(
    underlying_updates: &mut HashMap<Principal, (u128, u128, u128, Vec<Principal>)>,
) -> Principal {
    STATE.with(|state| {
        let state = state.borrow();
        let underlying_info = state.underlying.get(&asset).unwrap();

        // Fetch price from a decentralized oracle
        let oracle_price = fetch_oracle_price(&asset).await;
        validate_price(oracle_price, underlying_info.price); // Validate against local price

        let liquidate_amount = calculate_liquidate_amount(
            info.balance,
            oracle_price, // Use oracle price
            underlying_info.liquidate_rate,
            decimals,
        );
        // ...
    })
}
// Helper function to fetch price from a decentralized oracle
async fn fetch_oracle_price(asset: &Principal) -> u128 {
    // Call the oracle contract to get the price
    let oracle_price: u128 = ic_cdk::api::call::call(*ORACLE_CONTRACT, "get_price",
(asset,)).await.unwrap();
    oracle_price
}
// Helper function to validate the oracle price
fn validate_price(oracle_price: u128, local_price: u128) {
    let price_deviation = (oracle_price as i128 - local_price as i128).abs();
    if price_deviation > MAX_PRICE_DEVIATION {
        panic!("Price deviation too high: potential manipulation detected");
    }
}
```

CLP: **Fixed @ commit [9b46822](9b46822)**

VelintSec: **Resolved**

7.3.5 Liquidation Calculation Precision Issue

Description: The code contains a liquidation precision issue because it performs division after multiplication, which can lead to significant precision loss. Specifically, dividing by

10^(decimals + 2) after multiplying balance, price, and liquidate_rate can result in incorrect liquidation amounts, especially when dealing with large numbers or small decimal values. This can cause unfair liquidations or financial losses for users.

```rust
// calculate_liquidate_amount() in src/vault/src/time_task.rs
pub fn calculate_liquidate_amount(
    balance: u128,
    price: u128,
    liquidate_rate: u128,
    decimals: u8,
) -> Result<u128, Error> {
    // balance * price * liquidate_rate / 100 / 10 ^ decimals
    balance
        .checked_mul(price)
        .ok_or(Error::Overflow)?
        .checked_mul(liquidate_rate)
        .ok_or(Error::Overflow)?
        .checked_div(10u128.pow(u32::from(decimals + 2)))
        .ok_or(Error::Overflow)
}
```

Impact: The liquidation precision issue can result in:

- Incorrect Liquidation Amounts: Users may be liquidated for incorrect amounts due to precision loss, leading to unfair financial outcomes.
- Financial Loss: Precision errors can cause users to lose more funds than intended or prevent them from recovering the correct amount.
- Protocol Instability: Repeated precision issues can undermine trust in the protocol and destabilize its operations.

Recommended Mitigation: To mitigate the liquidation precision issue, ensure that calculations maintain sufficient precision throughout the process. Below are two recommended solutions with realistic code examples:

- Use higher precision arithmetic, perform calculations using higher precision arithmetic (e.g., u256 or fixed-point arithmetic) to avoid precision loss during intermediate steps.

```rust
use primitive_types::U256; // Use a higher-precision arithmetic library

pub fn calculate_liquidate_amount(
    balance: u128,
    price: u128,
    liquidate_rate: u128,
    decimals: u8,
) -> Result<u128, Error> {
    // Convert inputs to U256 for higher precision
    let balance = U256::from(balance);
    let price = U256::from(price);
    let liquidate_rate = U256::from(liquidate_rate);
    let divisor = U256::from(10).pow(U256::from(decimals + 2));

    // Perform the calculation with higher precision
    let result = balance
        .checked_mul(price)
        .ok_or(Error::Overflow)?
        .checked_mul(liquidate_rate)
        .ok_or(Error::Overflow)?
        .checked_div(divisor)
        .ok_or(Error::Overflow)?;

    // Convert back to u128
    if result > U256::from(u128::MAX) {
        return Err(Error::Overflow);
    }
    Ok(result.as_u128())
}
```

- Adjust the order of operations to minimize precision loss. For example, divide by 10^decimals before multiplying by liquidate_rate.

```
pub fn calculate_liquidate_amount(
    balance: u128,
    price: u128,
    liquidate_rate: u128,
    decimals: u8,
) -> Result<u128, Error> {
    // Divide by 10^decimals first to reduce the magnitude of intermediate results
    let scaled_balance = balance
        .checked_div(10u128.pow(u32::from(decimals)))
        .ok_or(Error::Overflow)?;

    // Perform the remaining calculations
    let result = scaled_balance
        .checked_mul(price)
        .ok_or(Error::Overflow)?
        .checked_mul(liquidate_rate)
        .ok_or(Error::Overflow)?
        .checked_div(100)
        .ok_or(Error::Overflow)?;

    Ok(result)
}
```

CLP: Fixed @ commit [9b46822]

VelintSec: Resolved

7.3.6 Numerical Overflow Risk

Description: The code contains an integer overflow vulnerability because it performs arithmetic operations (multiplication, subtraction, and addition) without proper checks. Specifically, the operations mul, sub, and add can overflow if the values involved are too large, leading to incorrect calculations and unexpected behavior.

```
// withdraw() in src/staking_pool/src/mining.rs
let new_reward = info
    .balance
    .mul(
        config
            .reward_per_token_stored
            .sub(info.reward_per_token_paid),
    )
    .div(MINING_MUL);
amount = info.reward.add(new_reward).add(amount);
```

Impact: The integer overflow vulnerability can result in:

- Incorrect Reward Calculations: Users may receive incorrect rewards due to overflow, leading to unfair distributions.
- Financial Loss: Attackers can exploit the overflow to manipulate reward calculations and drain funds from the protocol.
- Protocol Instability: Repeated overflow issues can undermine trust in the protocol and destabilize its operations.

Recommended Mitigation: To mitigate the integer overflow vulnerability, ensure that all arithmetic operations are performed safely, with proper checks or using libraries designed to prevent overflow. Below are two recommended solutions with realistic code examples:

- Replace raw arithmetic operations with a safe math library (Rust's checked_* methods) to prevent overflow.

```
use num_traits::CheckedSub; // For safe subtraction
use num_traits::CheckedMul; // For safe multiplication
use num_traits::CheckedAdd; // For safe addition

let new_reward = info
    .balance
    .checked_mul( // Safe multiplication
        config
            .reward_per_token_stored
            .checked_sub(info.reward_per_token_paid) // Safe subtraction
            .ok_or(Error::Overflow)?,
    )
    .ok_or(Error::Overflow)?
    .checked_div(MINING_MUL)
    .ok_or(Error::Overflow)?;

amount = info
    .reward
    .checked_add(new_reward) // Safe addition
    .ok_or(Error::Overflow)?
    .checked_add(amount) // Safe addition
    .ok_or(Error::Overflow)?;
```

- Use saturated arithmetic operations, which cap the result at the maximum or minimum value instead of overflowing.

```
let new_reward = info
        .balance
        .saturating_mul( // Saturated multiplication
                config
                        .reward_per_token_stored
                        .saturating_sub(info.reward_per_token_paid), // Saturated subtraction
        )
        .saturating_div(MINING_MUL); // Saturated division

amount = info
        .reward
        .saturating_add(new_reward) // Saturated addition
        .saturating_add(amount); // Saturated addition
```

CLP: **Fixed @ commit [9b46822]**

VelintSec: **Resolved**