Security Audit Report for Omnity

Date: June 26, 2024 **Version:** 1.0
Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Omnity Network
Target	Omnity

Version History

Version	Date	Description
1.0	June 26, 2024	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-06-26 13:53:50+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

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

The focus of this audit is on Omnity [1] of Omnity Network. Omnity is an omni-chain interoperability protocol built on the Internet Computer (IC) [2] specially designed to fit the modular blockchain landscape. It is implemented by a set of smart contracts deployed on IC and it currently supports BTC network to IC network and vice versa thanks to the native integrations with Bitcoin and Ethereum on IC. Its first launch is right after the 2024 Bitcoin Halving with its first settlement chain, Bitcoin, and first assets class, Runes [3].

Please note that this audit is limited to the smart contracts located within the `customs/bitcoin`, `hub` and `route/icp` folders of the repository. `tx.rs`, `signature.rs`, `address.rs`, `management.rs` in folder `customs/bitcoin` and files intended for test purposes are not within 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
Omnity	`Version 1`	`455d208533d51ce8d649f9337e43dd6210f4585e`
	`Version 2`	`d2360378775c79969d4242d56bb1fcb4669e6ee7`

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

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

[2] https://internetcomputer.org/

[3] https://rodarmor.com/blog/runes/

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.

1.3 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 [4] and Common Weakness Enumeration [5]. 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:

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

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

- **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.

Chapter 2 Findings

In total, we found **eleven** potential security issues. Besides, we have **four** recommendations and **nine** notes.

- High Risk: 2
- Medium Risk: 7
- Low Risk: 2
- Recommendation: 4
- Note: 9

ID	Severity	Description	Category	Status
1	Medium	Lack of popping skipped requests in function `build_batch()`	Software Security	Fixed
2	High	Infinite loop in the runes oracle	Software Security	Fixed
3	Medium	DoS attack by generating invalid tickets	DeFi Security	Fixed
4	High	DoS attack due to unremoved invalid tickets	DeFi Security	Fixed
5	Medium	DoS of redemption with dust runes	DeFi Security	Confirmed
6	Low	Metrics and logs can be publicly revealed	DeFi Security	Confirmed
7	Medium	Improper authentication in the hub	DeFi Security	Fixed
8	Low	Potential incorrect result returned from the function `fetch_main_utxos`	DeFi Security	Fixed
9	Medium	Tokens' destination chains can't be updated	Defi Security	Fixed
10	Medium	Lack of check on the transfer and redemption target	DeFi Security	Fixed
11	Medium	Lack of recovery in function `generate_ticket()`	Defi Security	Fixed
12	-	Typos in the contract	Recommendation	Confirmed
13	-	Redundant status `GenTicketStatus.Invalid`	Recommendation	Fixed
14	-	Redundant variable `btc_network`	Recommendation	Fixed
15	-	Redundant function `repub_2_subscribers()`	Recommendation	Fixed
16	-	Potential centralized risks	Note	-
17	-	Tickets are processed in the txid order	Note	-
18	-	Lack of cross-chain capability for multiple rune types or destinations in one Bitcoin transaction	Note	-
19	-	Potential temporary block of cross-chain requests due to deactivation of chains	Note	-
20	-	Lack of refunding mechanism for user's mistaken operations	Note	-
21	-	Inconsistency of cross-chain runes amount limitation	Note	-
22	-	Potential insufficient fees for Bitcoin resubmissions	Note	-
23	-	Potential insufficient cycles in upgrade	Note	-
24	-	Potential double spending by resubmitted tickets	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Lack of popping skipped requests in function `build_batch()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Before submitting pending requests, the bitcoin custom will build a batch with function `build_batch()`, which will skip the requests if the encoded script length is greater than 82 bytes.

However, `build_batch()` doesn't pop the requests out from the `edict` when it is skipped. In this case, the follow up requests may also be skipped since the script length is greater than 82 bytes.

```rust
/// Forms a batch of release_token requests that the customs can fulfill.
pub fn build_batch(&mut self, rune_id: RuneId, max_size: usize) -> Vec<ReleaseTokenRequest> {
    assert!(self.pending_release_token_requests.contains_key(&rune_id));


    let available_utxos_value = self
        .available_runes_utxos
        .iter()
        .filter(|u| u.runes.rune_id.eq(&rune_id))
        .map(|u| u.runes.amount)
        .sum::<u128>();
    let mut batch = vec![];
    let mut tx_amount = 0;
    let requests = self
        .pending_release_token_requests
        .entry(rune_id)
        .or_default();


    let mut edicts = vec![];
    for req in std::mem::take(requests) {
        edicts.push(Edict {
            id: req.rune_id.into(),
            amount: req.amount,
            output: 0,
        });
        // Maybe there is a better optimized version.
        let script = Runestone {
            edicts: edicts.clone(),
        }
        .encipher();
        if script.len() > 82
            || available_utxos_value < req.amount + tx_amount
            || batch.len() >= max_size
        {
            // Put this request back to the queue until we have enough liquid UTXOs.
            requests.push(req);
        } else {
            tx_amount += req.amount;
            batch.push(req.clone());
        }
    }


    batch
}
```

Listing 2.1: customs/bitcoin/src/state.rs

Impact Fewer requests can be handled during a submitting requests task.

Suggestion Pop the skipped request from `edicts`.

2.1.2 Infinite loop in the runes oracle

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The runes oracle processes pending requests from the beginning of a queue. However, an endless loop will occur if a request was previously decided as invalid. As shown in the code below, `pending_requests.front()` will keep getting the same invalid request from `pending_requests`.

```
while !self.pending_requests.is_empty() {
    let request = self.pending_requests.front().unwrap();
    if self.invalid_requests.contains(&request.txid) {
        continue;
}
```

Listing 2.2: customs/runes_oracle/src/executor.rs

Impact Users can't transfer runes from the `Bitcoin` chain to the target chain, and their runes will be locked.

Suggestion Remove the `invalid_requests` variable.

Note This issue is actually out of our audit scope. However, we reviewed part of the code while auditing the bitcoin custom module and this critical issue deserves to be reported.

2.2 DeFi Security

2.2.1 DoS attack by generating invalid tickets

Severity Medium

Status Fixed

Introduced by Version 1

Description According to the design, when users want to transfer runes from the Bitcoin chain to the target chain, they must first transfer the runes to the Bitcoin address specified by the bitcoin_customs canister, and then invoke the function generate_ticket() to generate a GenTicketRequest, which will be verified with function update_runes_balance(). Note that function generate_ticket() has no access control.

In this case, an attacker can monitor the Bitcoin transactions and invoke the function generate_ticket() before legitimate users with the same txid but incorrect parameters, causing a DoS attack.

```rust
pub async fn generate_ticket(args: GenerateTicketArgs) -> Result<(), GenerateTicketError> {
    if read_state(|s| s.chain_state == ChainState::Deactive) {
        return Err(GenerateTicketError::TemporarilyUnavailable(
            "chain state is deactive!".into(),
        ));
    }


    init_ecdsa_public_key().await;
    let _guard = generate_ticket_guard()?;


    let rune_id = RuneId::from_str(&args.rune_id)
        .map_err(|e| GenerateTicketError::InvalidRuneId(e.to_string()))?;


    let txid = Txid::from_str(&args.txid).map_err(|_| GenerateTicketError::InvalidTxId)?;


    if !read_state(|s| {
        s.counterparties
            .get(&args.target_chain_id)
            .is_some_and(|c| c.chain_state == ChainState::Active)
    }) {
        return Err(GenerateTicketError::UnsupportedChainId(
            args.target_chain_id.clone(),
        ));
    }


    let token_id = read_state(|s| {
        if let Some((token_id, _)) = s.tokens.iter().find(|(_, (r, _))| rune_id.eq(r)) {
            Ok(token_id.clone())
        } else {
            Err(GenerateTicketError::UnsupportedToken(args.rune_id))
        }
    })?;


    read_state(|s| match s.generate_ticket_status(txid) {
        GenTicketStatus::Pending(_) => Err(GenerateTicketError::AlreadySubmitted),
        GenTicketStatus::Invalid | GenTicketStatus::Finalized => {
            Err(GenerateTicketError::AleardyProcessed)
        }
        GenTicketStatus::Unknown => Ok(()),
    })?;


    let (btc_network, min_confirmations) = read_state(|s| (s.btc_network, s.min_confirmations))
        ;


    let destination = Destination {
        target_chain_id: args.target_chain_id.clone(),
        receiver: args.receiver.clone(),
        token: None,
    };


    let address = read_state(|s| destination_to_p2wpkh_address_from_state(s, &destination));


    // In order to prevent the memory from being exhausted,
    // ensure that the user has transferred token to this address.
    let utxos = get_utxos(btc_network, &address, min_confirmations, CallSource::Client)
        .await
        .map_err(|call_err| {
            GenerateTicketError::TemporarilyUnavailable(format!(
                "Failed to call bitcoin canister: {}",
                call_err
            ))
        })?
        .utxos;


    let new_utxos = read_state(|s| s.new_utxos(utxos, Some(txid)));
    if new_utxos.len() == 0 {
        return Err(GenerateTicketError::NoNewUtxos);
    }


    let request = GenTicketRequest {
        address,
        target_chain_id: args.target_chain_id,
        receiver: args.receiver,
        token_id,
        rune_id,
        amount: args.amount,
        txid,
        received_at: ic_cdk::api::time(),
    };


    mutate_state(|s| {
```

```
        audit::accept_generate_ticket_request(s, request);
        audit::add_utxos(s, destination, new_utxos, true);
    });
    Ok(())
}
```

Listing 2.3: customs/bitcoin/src/updates/generate_ticket.rs

Impact Users can't transfer runes from the `Bitcoin` chain to the target chain, and their runes will be locked.

Suggestion Add access controls on the function `generate_ticket()`.

Feedback from the project An on-chain runes indexer will be used and users won't submit amounts.

Note The attack can keep calling `generate_ticket()` with invalid amounts before the on-chain runes indexer is used.

2.2.2 DoS attack due to unremoved invalid tickets

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Oracle first invokes the function `get_pending_gen_ticket_requests()` to retrieve tickets and then updates the balance by calling `update_runes_balance()`. Each time, at most 50 ones sorted by the `Bitcoin txid` can be retrieved and processed.

However, the function `update_runes_balance()` won't remove invalid pending tickets (e.g., the amount doesn't match), which will be retrieved again next time. In this case, attackers can generate many (e.g., 50) tickets with small `txid` (pre-computable) but invalid amounts. Consequently, the invalid tickets will always be retrieved by the oracle, and the runes balance cannot be updated.

```
pub async fn start(&mut self) {
    let ticker = Ticker::new(0.., Duration::from_secs(60));
    for _ in ticker {
        if self.pending_requests.is_empty() {
            match self.customs.get_pending_gen_ticket_requests(None, 50).await {
                Ok(requests) => requests
                    .iter()
                    .for_each(|r| self.pending_requests.push_back(r.clone())),
                Err(err) => {
                    log::error!("failed to get pending requests: {}", err);
                    continue;
                }
            }
        }
        while !self.pending_requests.is_empty() {
            let request = self.pending_requests.front().unwrap();
            if self.invalid_requests.contains(&request.txid) {
                continue;
            }


            match self.indexer.get_transaction(request.txid).await {
                Ok(tx) => {
                    let mut balances = tx.get_runes_balances();
                    balances.retain(|b| {
                        b.address == request.address && b.rune_id == request.rune_id.to_string()
                    });


                    match self
                        .customs
                        .update_runes_balance(
                            request.txid,
                            balances
                                .iter()
                                .map(|b| {
                                    let rune_id = RuneId::from_str(&b.rune_id).unwrap();
                                    state::RunesBalance {
                                        rune_id,
                                        vout: b.vout,
                                        amount: b.amount,
                                    }
                                })
                                .collect(),
                        )
                        .await
                    {
                        Ok(result) => match result {
                            Ok(()) => {
                                log::info!(
                                    "update runes balance success for txid:{}",
                                    request.txid
                                );
                            }
                            Err(UpdateRunesBalanceError::AleardyProcessed) => {}
                            Err(UpdateRunesBalanceError::RequestNotFound) => {
                                // Should never happen.
                                log::error!("request not found for txid:{}", request.txid);
                            }
                            Err(UpdateRunesBalanceError::MismatchWithGenTicketReq) => {
                                self.invalid_requests.insert(request.txid);
                                log::error!(
                                    "mismatch with ticket request for txid:{}",
                                    request.txid
                                );
                            }
                            Err(UpdateRunesBalanceError::UtxoNotFound) => {
                                // Should never happen.
                                log::error!("utxo not found for txid:{}", request.txid);
                            }
                            Err(UpdateRunesBalanceError::SendTicketErr(err)) => {
                                    log::error!(
                                        "send ticket err({}) for txid:{}",
                                        err,
                                        request.txid
                                    );
                                }
                            },
                            Err(err) => {
                                log::error!("failed to update runes balance: {}", err);
                                break;
                            }
                        }
                    }
                    Err(err) => {
                        log::error!("failed to get transaction from indexer: {:?}", err);
                        break;
                    }
                }
                self.pending_requests.pop_front();
            }
        }
    }
}
```

Listing 2.4: customs/runes_oracle/src/executor.rs

```
#[query]
fn get_pending_gen_ticket_requests(args: GetGenTicketReqsArgs) -> Vec<GenTicketRequest> {
    let start = args.start_txid.map_or(Unbounded, |txid| Excluded(txid));
    let count = max(50, args.max_count) as usize;
    read_state(|s| {
        s.pending_gen_ticket_requests
            .range((start, Unbounded))
            .take(count)
            .map(|(_, req)| req.clone())
            .collect()
    })
}
```

Listing 2.5: customs/bitcoin/src/main.rs

```
pub async fn update_runes_balance(
    args: UpdateRunesBalanceArgs,
 ) -> Result<(), UpdateRunesBalanceError> {
    for balance in &args.balances {
        let outpoint = OutPoint {
            txid: args.txid,
            vout: balance.vout,
        };
        read_state(|s| match s.outpoint_destination.get(&outpoint) {
            Some(_) => Ok(()),
            None => Err(UpdateRunesBalanceError::UtxoNotFound),
        })?;
    }


    let req = read_state(|s| match s.generate_ticket_status(args.txid) {
        GenTicketStatus::Invalid | GenTicketStatus::Finalized => {
            Err(UpdateRunesBalanceError::AleardyProcessed)
        }
        GenTicketStatus::Unknown => Err(UpdateRunesBalanceError::RequestNotFound),
        GenTicketStatus::Pending(req) => Ok(req),
    })?;


    let amount = args.balances.iter().map(|b| b.amount).sum::<u128>();
    if amount != req.amount || args.balances.iter().any(|b| b.rune_id != req.rune_id) {
        return Err(UpdateRunesBalanceError::MismatchWithGenTicketReq);
    }
```

Listing 2.6: customs/bitcoin/src/updates/update_runes_balance.rs

Impact Users can't transfer runes from the `Bitcoin` chain to the target chain, and their runes will be locked.

Suggestion Remove invalid tickets in the function `update_runes_balance()`.

2.2.3 DoS of redemption with dust runes

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The `Bitcoin` custom handles pending redemption requests every 5 seconds. For each type of rune, it tries to find the smallest set of utxo where the sum of available runes is bigger than the redeemed amount, and the remaining runes will be transferred to a main `Bitcoin` address after they are finalized.

However, the finalized time is around 1-2 hours, which is much longer than the interval time of handling redeem tickets. In this case, attackers can redeem dust runes to form small redeem batches and make the remaining runes unavailable until the redemption requests are finalized.

```
 let confirmed_transactions: Vec<_> =
 state::read_state(|s| finalized_txs(&s.submitted_transactions, &new_runes_utxos));


// It's possible that some transactions we considered lost or rejected became finalized in the
// meantime. If that happens, we should stop waiting for replacement transactions to finalize.
let unstuck_transactions: Vec<_> =
  state::read_state(|s| finalized_txs(&s.stuck_transactions, &new_runes_utxos));


state::mutate_state(|s| {
  let btc_utxos = get_btc_utxos_from_confirmed_tx(&confirmed_transactions);
  audit::add_utxos(s, main_btc_destination.clone(), btc_utxos, false);

```

```

    for (dest, utxos) in dest_runes_utxos {
        audit::add_utxos(s, dest, utxos, true);
    }
    for tx in &confirmed_transactions {
        state::audit::confirm_transaction(s, &tx.txid);
        let balance = RunesBalance {
            rune_id: tx.runes_change_output.rune_id.clone(),
            vout: tx.runes_change_output.vout,
            amount: tx.runes_change_output.value,
        };
        audit::update_runes_balance(s, tx.txid, balance);
        maybe_finalized_transactions.remove(&tx.txid);
    }
});


for tx in &unstuck_transactions {
    state::read_state(|s| {
        if let Some(replacement_txid) = s.find_last_replacement_tx(&tx.txid) {
            maybe_finalized_transactions.remove(replacement_txid);
        }
    });
}


state::mutate_state(|s| {
    let btc_utxos = get_btc_utxos_from_confirmed_tx(&unstuck_transactions);
    audit::add_utxos(s, main_btc_destination, btc_utxos, false);
    for tx in unstuck_transactions {
        log!(
            P0,
            "[finalize_requests]: finalized transaction {} assumed to be stuck",
            &tx.txid
        );
        state::audit::confirm_transaction(s, &tx.txid);
        let balance = RunesBalance {
            rune_id: tx.runes_change_output.rune_id.clone(),
            vout: tx.runes_change_output.vout,
            amount: tx.runes_change_output.value,
        };
        audit::update_runes_balance(s, tx.txid, balance);
    }
});
```

Listing 2.7: customs/bitcoin/src/lib.rs

Impact Most of the runes are unavailable for redemption requests.

Suggestion Increase the interval time of function `process_tx_task()` and give priority to redemption requests with larger amounts.

Feedback from the project The redeem transaction requires the user to pay the gas fee, and customs will also batch requests in one transaction.

2.2.4 Metrics and logs can be publicly revealed

Severity Low

Status Confirmed

Introduced by `Version 1`

Description According to the ICP security best practices [1], cycle balance should not be publicly revealed:

Publicly revealing the canister's cycles balance allows an attacker to measure the number of instructions spent by executing the canister methods on the attacker's input. Then the attacker might be able to learn which code paths were taken during execution and derive secret information based on that. Moreover, the attacker can learn which methods and their inputs consume a lot of cycles to mount a cycles draining attack.

However, anyone can call functions `http_request()` , `get_logs()` , and `get_log_records()` , revealing metrics like the canisters' cycle balance.

Impact Canisters are vulnerable to cycle-draining attacks.

Suggestion Ensure metric and log functions can only be called by the authorities.

2.2.5 Improper authentication in the hub

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description When the proposal of adding a new chain is executed in the `hub` canister, the new chain's `canister_id` and `chain_id` are inserted into the HashMap `authorized_caller` as key and value. This means that the new chain is granted the `auth` role, and the `auth` role in the `hub` has the permission to propose and execute proposals, allowing the chain to modify the state (activate and deactivate) of other chains. This permission hierarchy is incorrect.

```rust
pub async fn execute_proposal(proposals: Vec<Proposal>) -> Result<(), Error> {
    for proposal in proposals.into_iter() {
        match proposal {
            Proposal::AddChain(chain_meta) => {
                // save new chain
                with_state_mut(|hub_state| {
                    info!(" save new chain: {:?}", chain_meta);
                    hub_state.add_chain(chain_meta.clone())
                })?;
                // publish directive for the new chain)
                info!(
                    "publish directive for ‘AddChain‘ proposal :{:?}",
                    chain_meta.to_string()
                );
                with_state_mut(|hub_state| {
                    let target_subs = chain_meta.counterparties.clone().unwrap_or_default();
```

[1] https://internetcomputer.org/docs/current/developer-docs/security/rust-canister-development-security-best-practices

```
                hub_state
                    .pub_directive(Some(target_subs), &Directive::AddChain(chain_meta.into())
                        )
            })?;
        }


        Proposal::AddToken(token_meata) => {
            info!(
                "publish directive for ‘AddToken‘ proposal :{:?}",
                token_meata
            );


            with_state_mut(|hub_state| {
                // save token info
                hub_state.add_token(token_meata.clone())?;
                // publish directive
                hub_state.pub_directive(
                    Some(token_meata.dst_chains.clone()),
                    &Directive::AddToken(token_meata.into()),
                )
            })?
        }


        Proposal::ToggleChainState(toggle_status) => {
            info!(
                "publish directive for ‘ToggleChainState‘ proposal :{:?}",
                toggle_status
            );


            with_state_mut(|hub_state| {
                // publish directive
                hub_state
                    .pub_directive(None, &Directive::ToggleChainState(toggle_status.clone()))
                        ?;
                // update dst chain state
                hub_state.update_chain_state(&toggle_status)
            })?;
        }


        Proposal::UpdateFee(factor) => {
            info!("publish directive for ‘UpdateFee‘ proposal :{:?}", factor);
            with_state_mut(|hub_state| {
                hub_state.update_fee(factor.clone())?;
                let target_subs = match &factor {
                    Factor::UpdateTargetChainFactor(factor) => {
                        hub_state.get_chains_by_counterparty(factor.target_chain_id.clone())
                    }
                    Factor::UpdateFeeTokenFactor(factor) => {
                            hub_state.get_chains_by_fee_token(factor.fee_token.clone())
                        }
                    };
                    hub_state
                        .pub_directive(Some(target_subs), &Directive::UpdateFee(factor.clone()))
                })?;
            }
        }
    }
    Ok(())
}
```

Listing 2.8: hub/src/proposal.rs

```
pub fn add_chain(&mut self, chain: ChainMeta) -> Result<(), Error> {
    // save chain
    self.chains
        .insert(chain.chain_id.to_string(), chain.clone());
    // update auth
    self.authorized_caller
        .insert(chain.canister_id.to_string(), chain.chain_id.to_string());
    record_event(&Event::AddedChain(chain.clone()));


    // update counterparties
    if let Some(counterparties) = chain.counterparties {
        counterparties.iter().try_for_each(|counterparty| {
            //check and update counterparty of dst chain
            self.update_chain_counterparties(counterparty, &chain.chain_id)
        })?;
    }


    Ok(())
}
```

Listing 2.9: hub/src/state.rs

```
pub fn auth() -> Result<(), String> {
    let caller = ic_cdk::api::caller();
    info!("auth for caller: {:?}", caller.to_string());
    with_state(|s| {
        if s.admin != caller
            && !ic_cdk::api::is_controller(&caller)
            && !s.authorized_caller.contains_key(&caller.to_string())
        {
            Err("Unauthorized!".into())
        } else {
            Ok(())
        }
    })
}
```

Listing 2.10: hub/src/state.rs

```
#[query(guard = "auth")]
pub async fn validate_proposal(proposals: Vec<Proposal>) -> Result<Vec<String>, Error> {
   proposal::validate_proposal(&proposals).await
}
#[update(guard = "auth")]
pub async fn execute_proposal(proposals: Vec<Proposal>) -> Result<(), Error> {
   proposal::execute_proposal(proposals).await
}
```

Listing 2.11: hub/src/service.rs

Impact Chains' routes are able to activate/deactivate other chains.

Suggestion Implement correct logic for authentication.

2.2.6 Potential incorrect result returned from the function `fetch_main_utxos`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the current implementation, the function `fetch_main_utxos` in the `bitcoin_custom` canister invokes the function `bitcoin_get_utxos()` of the management canister for each address. The function `fetch_main_utxos` constructs a `BTreeMap` structure to record all unknown UTXOs related to each destination.

However, when the cross-canister call `bitcoin_get_utxos()` fails with `CallError`, the function directly returns an empty result as `BTreeMap::default()` on Line 174 instead of including the results already correctly handled or continuing to process the remaining addresses.

```
151  async fn fetch_main_utxos(
152      addresses: Vec<(Destination, BitcoinAddress)>,
153      btc_network: Network,
154      min_confirmations: u32,
155   ) -> BTreeMap<Destination, Vec<Utxo>> {
156      let mut result = BTreeMap::default();
157      for (main_dest, main_address) in addresses {
158          let utxos = match management::get_utxos(
159              btc_network,
160              &main_address.display(btc_network),
161              min_confirmations,
162              management::CallSource::Custom,
163          )
164          .await
165          {
166              Ok(response) => response.utxos,
167              Err(e) => {
168                  log!(
169                      P0,
170                      "[fetch_main_utxos]: failed to fetch UTXOs for the main address {}: {}",
171                      main_address.display(btc_network),
172                      e
```

```rust
                );
                return BTreeMap::default();
            }
        };


        result.insert(
            main_dest.clone(),
            state::read_state(|s| match s.utxos_state_destinations.get(&main_dest) {
                Some(known_utxos) => utxos
                    .into_iter()
                    .filter(|u| !known_utxos.contains(u))
                    .collect(),
                None => utxos,
            }),
        );
    }
    result
}
```

Listing 2.12: customs/bitcoin/src/lib.rs

Impact A single `CallError` can revert the entire `fetch_main_utxos` and return an empty result.

Suggestion Revise the code to correctly handle `CallError` and proceed with the remaining addresses.

2.2.7 Tokens' destination chains can't be updated

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description When the hub executes an `AddToken` proposal, it will publish the directives for all the destination chains in the token metadata. However, the specified destination chains can't be updated. In this case, when a new chain is added, it can't use an existing token. This is because it doesn't have the directive and a new `AddToken` proposal will fail in the function `validate_proposal()` with `TokenAlreadyExisting` error.

```rust
pub async fn validate_proposal(proposals: &Vec<Proposal>) -> Result<Vec<String>, Error> {
    if proposals.is_empty() {
        return Err(Error::ProposalError(
            "Proposal can not be empty".to_string(),
        ));
    }
    let mut proposal_msgs = Vec::new();
    for proposal in proposals.iter() {
        match proposal {
            Proposal::AddChain(chain_meta) => {
                if chain_meta.chain_id.is_empty() {
                    return Err(Error::ProposalError(
                        "Chain name can not be empty".to_string(),
                    ));
                }


                if matches!(chain_meta.chain_state, ChainState::Deactive) {
                    return Err(Error::ProposalError(
                        "The status of the new chain state must be active".to_string(),
                    ));
                }


                with_state(|hub_state| {
                    hub_state.chain(&chain_meta.chain_id).map_or(Ok(()), |_| {
                        Err(Error::ChainAlreadyExisting(chain_meta.chain_id.to_string()))
                    })
                })?;


                proposal_msgs.push(format!("Tne AddChain proposal: {}", chain_meta));
            }
            Proposal::AddToken(token_meta) => {
                if token_meta.token_id.is_empty()
                    || token_meta.symbol.is_empty()
                    || token_meta.issue_chain.is_empty()
                {
                    return Err(Error::ProposalError(
                        "Token id, token symbol or issue chain can not be empty".to_string(),
                    ));
                }
                with_state(|hub_state| {
                    // check token repetitive
                    hub_state.token(&token_meta.token_id).map_or(Ok(()), |_| {
                        Err(Error::TokenAlreadyExisting(token_meta.to_string()))
                    })?;
```

Listing 2.13: hub/src/proposal.rs

Impact New chains can't use existing tokens.

Suggestion Add a token update method in the hub.

2.2.8 Lack of check on the transfer and redemption target

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description Currently, the hub allows tokens to be transferred to issue chains and redeemed to non-issue chains, which is incorrect.

```
pub fn check_and_update(&mut self, ticket: &Ticket) -> Result<(), Error> {
    // check ticket id repetitive
    if self.cross_ledger.contains_key(&ticket.ticket_id) {
        return Err(Error::AlreadyExistingTicketId(ticket.ticket_id.to_string()));
    }
    // check chain and state
    self.available_chain(&ticket.src_chain)?;
    self.available_chain(&ticket.dst_chain)?;


    //parse ticket token amount to unsigned bigint
    let ticket_amount: u128 = ticket.amount.parse().map_err(|e: ParseIntError| {
        Error::TicketAmountParseError(ticket.amount.to_string(), e.to_string())
    })?;


    // check token on chain availability
    match ticket.action {
        TxAction::Transfer => {
            // ticket from issue chain
            if self.is_origin(&ticket.src_chain, &ticket.token)? {
                info!(
                    "ticket token({}) from issue chain({}).",
                    ticket.token, ticket.src_chain,
                );


                // just update token amount on dst chain
                self.add_token_position(
                    TokenKey::from(ticket.dst_chain.to_string(), ticket.token.to_string()),
                    ticket_amount,
                )?;


            // not from issue chain
            } else {
                info!(
                    "ticket token({}) from a not issue chain({}).",
                    ticket.token, ticket.src_chain,
                );


                // update token amount on src chain
                self.update_token_position(
                    TokenKey::from(ticket.src_chain.to_string(), ticket.token.to_string()),
                    |total_amount| {
                        // check src chain token balance
                        if *total_amount < ticket_amount {
                            return Err::<u128, Error>(Error::NotSufficientTokens(
                                ticket.token.to_string(),
                                ticket.src_chain.to_string(),
                            ));
                        }
                        *total_amount -= ticket_amount;
                        Ok(*total_amount)
                    },
                )?;
                // update token amount on dst chain
                self.add_token_position(
                    TokenKey::from(ticket.dst_chain.to_string(), ticket.token.to_string()),
                    ticket_amount,
                )?;
            }
        }


        TxAction::Redeem => {
            // update token amount on src chain
            self.update_token_position(
                TokenKey::from(ticket.src_chain.to_string(), ticket.token.to_string()),
                |total_amount| {
                    // check src chain token balance
                    if *total_amount < ticket_amount {
                        return Err::<u128, Error>(Error::NotSufficientTokens(
                            ticket.token.to_string(),
                            ticket.src_chain.to_string(),
                        ));
                    }
                    *total_amount -= ticket_amount;
                    Ok(*total_amount)
                },
            )?;


            //  if the dst chain is not issue chain,then update token amount on dst chain
            if !self.is_origin(&ticket.dst_chain, &ticket.token)? {
                self.update_token_position(
                    TokenKey::from(ticket.dst_chain.to_string(), ticket.token.to_string()),
                    |total_amount| {
                        *total_amount += ticket_amount;
                        Ok(*total_amount)
                    },
                )?;
            }
        }
    }


    Ok(())
}
```

Listing 2.14: hub/src/state.rs

Impact Tokens can be transferred to an issue chain or redeemed to a non-issue chain.

Suggestion The function `check_and_update()` should only allow transfer actions to non-issue chains and redeem actions to issue chains.

Feedback from the project Transfer to issue chain will fail. Redeeming to a non-issue chain is by-design.

2.2.9 Lack of recovery in function `generate_ticket()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `generate_ticket()` of canister route, there is no recovery mechanism to refund the redeem fees and burned tokens if `send_ticket()` fails, which doesn't follow the best security practice [2] and will lead to user's assets loss.

```rust
pub async fn generate_ticket(
    req: GenerateTicketReq,
 ) -> Result<GenerateTicketOk, GenerateTicketError> {
    if read_state(|s| s.chain_state == ChainState::Deactive) {
        return Err(GenerateTicketError::TemporarilyUnavailable(
            "chain state is deactive!".into(),
        ));
    }


    if !read_state(|s| {
        s.counterparties
            .get(&req.target_chain_id)
            .is_some_and(|c| c.chain_state == ChainState::Active)
    }) {
        return Err(GenerateTicketError::UnsupportedChainId(
            req.target_chain_id.clone(),
        ));
    }


    let ledger_id = read_state(|s| match s.token_ledgers.get(&req.token_id) {
        Some(ledger_id) => Ok(ledger_id.clone()),
        None => Err(GenerateTicketError::UnsupportedToken(req.token_id.clone())),
    })?;


    charge_redeem_fee(caller(), &req.target_chain_id).await?;


    let caller = ic_cdk::caller();
    let user = Account {
        owner: caller,
        subaccount: req.from_subaccount,
    };


    let block_index = burn_token_icrc2(ledger_id, user, req.amount).await?;
    let ticket_id = format!("{}_{}", ledger_id.to_string(), block_index.to_string());

```

[2] https://internetcomputer.org/docs/current/developer-docs/security/rust-canister-development-security-best-practices

```

    let (hub_principal, chain_id) = read_state(|s| (s.hub_principal, s.chain_id.clone()));
    hub::send_ticket(
        hub_principal,
        Ticket {
            ticket_id: ticket_id.clone(),
            ticket_type: omnity_types::TicketType::Normal,
            ticket_time: ic_cdk::api::time(),
            src_chain: chain_id,
            dst_chain: req.target_chain_id.clone(),
            action: TxAction::Redeem,
            token: req.token_id.clone(),
            amount: req.amount.to_string(),
            sender: None,
            receiver: req.receiver.clone(),
            memo: None,
        },
    )
    .await
    .map_err(|err| GenerateTicketError::SendTicketErr(format!("{}", err)))?;


    audit::finalize_gen_ticket(ticket_id.clone(), req);
    Ok(GenerateTicketOk { ticket_id })
}
```

Listing 2.15: route/icp/src/updates/generate_ticket.rs

Impact Users lose tokens and fees if route's `send_ticket()` fails.

Suggestion Implement related recovery logic if `send_ticket()` returns an error.

2.3 Additional Recommendation

2.3.1 Typos in the contract

Status Confirmed

Introduced by `Version 1`

Description There are some typos in the project, such as the `AleardyProcessed` in `customs/updates/generate_ticket.rs` and `customs/updates/update_runes_balance.rs`, the `chagne` in the `customs/lib.rs`, the `exection` and `checke` in `hub/service.rs`.

```
pub enum GenerateTicketError {
    TemporarilyUnavailable(String),
    AlreadySubmitted,
    AleardyProcessed,
```

Listing 2.16: customs/updates/generate_ticket.rs

```
pub enum UpdateRunesBalanceError {
    RequestNotFound,
    AleardyProcessed,
```

Listing 2.17: customs/updates/update_runes_balance.rs

```
// Additional MIN_OUTPUT_AMOUNT are used as the value of the outputs(two chagne output +
    multiple dest runes outputs).
```

Listing 2.18: customs/lib.rs

```
// exection proposal and generate directives
```

Listing 2.19: hub/service.rs

```
// checke ticket and update token on chain
```

Listing 2.20: hub/services.rs

Suggestion Revise the typos.

Note The typos are fixed except the `exection` one.

2.3.2 Redundant status `GenTicketStatus.Invalid`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the current implementation, the status `Invalid` in `GenTicketStatus` is never used. Therefore, the check of corresponding status is redundant.

```
pub async fn generate_ticket(args: GenerateTicketArgs) -> Result<(), GenerateTicketError> {
    if read_state(|s| s.chain_state == ChainState::Deactive) {
        return Err(GenerateTicketError::TemporarilyUnavailable(
            "chain state is deactive!".into(),
        ));
    }


    init_ecdsa_public_key().await;
    let _guard = generate_ticket_guard()?;


    let rune_id = RuneId::from_str(&args.rune_id)
        .map_err(|e| GenerateTicketError::InvalidRuneId(e.to_string()))?;


    let txid = Txid::from_str(&args.txid).map_err(|_| GenerateTicketError::InvalidTxId)?;


    if !read_state(|s| {
        s.counterparties
            .get(&args.target_chain_id)
            .is_some_and(|c| c.chain_state == ChainState::Active)
    }) {
        return Err(GenerateTicketError::UnsupportedChainId(
            args.target_chain_id.clone(),
        ));
    }


    let token_id = read_state(|s| {
        if let Some((token_id, _)) = s.tokens.iter().find(|(_, (r, _))| rune_id.eq(r)) {
            Ok(token_id.clone())
        } else {
            Err(GenerateTicketError::UnsupportedToken(args.rune_id))
        }
    })?;


    read_state(|s| match s.generate_ticket_status(txid) {
        GenTicketStatus::Pending(_) => Err(GenerateTicketError::AlreadySubmitted),
        GenTicketStatus::Invalid | GenTicketStatus::Finalized => {
            Err(GenerateTicketError::AleardyProcessed)
        }
        GenTicketStatus::Unknown => Ok(()),
    })?;


    let (btc_network, min_confirmations) = read_state(|s| (s.btc_network, s.min_confirmations))
        ;


    let destination = Destination {
        target_chain_id: args.target_chain_id.clone(),
        receiver: args.receiver.clone(),
        token: None,
    };


    let address = read_state(|s| destination_to_p2wpkh_address_from_state(s, &destination));


    // In order to prevent the memory from being exhausted,
    // ensure that the user has transferred token to this address.
    let utxos = get_utxos(btc_network, &address, min_confirmations, CallSource::Client)
        .await
        .map_err(|call_err| {
            GenerateTicketError::TemporarilyUnavailable(format!(
                "Failed to call bitcoin canister: {}",
                call_err
            ))
        })?
        .utxos;


    let new_utxos = read_state(|s| s.new_utxos(utxos, Some(txid)));
    if new_utxos.len() == 0 {
        return Err(GenerateTicketError::NoNewUtxos);
    }


    let request = GenTicketRequest {
        address,
        target_chain_id: args.target_chain_id,
        receiver: args.receiver,
        token_id,
        rune_id,
        amount: args.amount,
        txid,
        received_at: ic_cdk::api::time(),
    };


    mutate_state(|s| {
        audit::accept_generate_ticket_request(s, request);
        audit::add_utxos(s, destination, new_utxos, true);
    });
    Ok(())
}
```

Listing 2.21: customs/bitcoin/src/updates/generate_ticket.rs

```
pub async fn update_runes_balance(
    args: UpdateRunesBalanceArgs,
) -> Result<(), UpdateRunesBalanceError> {
    for balance in &args.balances {
        let outpoint = OutPoint {
            txid: args.txid,
            vout: balance.vout,
        };
        read_state(|s| match s.outpoint_destination.get(&outpoint) {
            Some(_) => Ok(()),
            None => Err(UpdateRunesBalanceError::UtxoNotFound),
        })?;
    }


    let req = read_state(|s| match s.generate_ticket_status(args.txid) {
        GenTicketStatus::Invalid | GenTicketStatus::Finalized => {
            Err(UpdateRunesBalanceError::AleardyProcessed)
        }
        GenTicketStatus::Unknown => Err(UpdateRunesBalanceError::RequestNotFound),
        GenTicketStatus::Pending(req) => Ok(req),
    })?;


    let amount = args.balances.iter().map(|b| b.amount).sum::<u128>();
    if amount != req.amount || args.balances.iter().any(|b| b.rune_id != req.rune_id) {
        return Err(UpdateRunesBalanceError::MismatchWithGenTicketReq);
    }


    let (hub_principal, chain_id) = read_state(|s| (s.hub_principal, s.chain_id.clone()));
    hub::send_ticket(
        hub_principal,
        Ticket {
            ticket_id: args.txid.to_string(),
            ticket_type: TicketType::Normal,
            ticket_time: ic_cdk::api::time(),
            src_chain: chain_id,
            dst_chain: req.target_chain_id.clone(),
            action: TxAction::Transfer,
            token: req.token_id.clone(),
            amount: req.amount.to_string(),
            sender: None,
            receiver: req.receiver.clone(),
            memo: None,
        },
    )
    .await
    .map_err(|err| UpdateRunesBalanceError::SendTicketErr(format!("{}", err)))?;


    mutate_state(|s| audit::finalize_ticket_request(s, &req, args.balances));


    Ok(())
}
```

Listing 2.22: customs/updates/update_runes_balance.rs

Suggestion Remove the redundant `Invalid` status.

2.3.3 Redundant variable `btc_network`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `btc_network` variable is fetched twice in the function `finalize_requests()`.

```
  let (btc_network, min_confirmations) =
  state::read_state(|s| (s.btc_network, s.min_confirmations));


let dest_runes_utxos =
  fetch_main_utxos(main_runes_addresses.clone(), btc_network, min_confirmations).await;


let new_runes_utxos = dest_runes_utxos
  .iter()
  .map(|(_, utxos)| utxos)
  .flatten()
  .map(|u| u.clone())
  .collect::<Vec<Utxo>>();
```

```


// Transactions whose change outpoint is present in the newly fetched UTXOs
// can be finalized. Note that all new customs transactions must have a
// change output because customs always charges a fee for converting tokens.
let confirmed_transactions: Vec<_> =
  state::read_state(|s| finalized_txs(&s.submitted_transactions, &new_runes_utxos));


// It's possible that some transactions we considered lost or rejected became finalized in the
// meantime. If that happens, we should stop waiting for replacement transactions to finalize.
let unstuck_transactions: Vec<_> =
  state::read_state(|s| finalized_txs(&s.stuck_transactions, &new_runes_utxos));


state::mutate_state(|s| {
  let btc_utxos = get_btc_utxos_from_confirmed_tx(&confirmed_transactions);
  audit::add_utxos(s, main_btc_destination.clone(), btc_utxos, false);


  for (dest, utxos) in dest_runes_utxos {
      audit::add_utxos(s, dest, utxos, true);
  }
  for tx in &confirmed_transactions {
      state::audit::confirm_transaction(s, &tx.txid);
      let balance = RunesBalance {
          rune_id: tx.runes_change_output.rune_id.clone(),
          vout: tx.runes_change_output.vout,
          amount: tx.runes_change_output.value,
      };
      audit::update_runes_balance(s, tx.txid, balance);
      maybe_finalized_transactions.remove(&tx.txid);
  }
});


for tx in &unstuck_transactions {
  state::read_state(|s| {
      if let Some(replacement_txid) = s.find_last_replacement_tx(&tx.txid) {
          maybe_finalized_transactions.remove(replacement_txid);
      }
  });
}


state::mutate_state(|s| {
  let btc_utxos = get_btc_utxos_from_confirmed_tx(&unstuck_transactions);
  audit::add_utxos(s, main_btc_destination, btc_utxos, false);
  for tx in unstuck_transactions {
      log!(
          P0,
          "[finalize_requests]: finalized transaction {} assumed to be stuck",
          &tx.txid
```

```
    );
    state::audit::confirm_transaction(s, &tx.txid);
    let balance = RunesBalance {
        rune_id: tx.runes_change_output.rune_id.clone(),
        vout: tx.runes_change_output.vout,
        amount: tx.runes_change_output.value,
    };
    audit::update_runes_balance(s, tx.txid, balance);
  }
});


// Do not replace transactions if less than MIN_RESUBMISSION_DELAY passed since their
// submission. This strategy works around short-term fee spikes.
maybe_finalized_transactions
  .retain(|_txid, tx| tx.submitted_at + MIN_RESUBMISSION_DELAY.as_nanos() as u64 <= now);


if maybe_finalized_transactions.is_empty() {
  // There are no transactions eligible for replacement.
  return;
}


let btc_network = state::read_state(|s| s.btc_network);
```

Listing 2.23: customs/bitcoin/src/lib.rs

Suggestion Remove the redundant variable `btc_network`.

2.3.4 Redundant function `repub_2_subscribers()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Currently, the `repub_2_subscribers()` function is not used. Additionally, the implementation executes all the directives in `self.directives`, which may cause the target chain to receive incorrect directives. This function should be removed.

```
  pub fn repub_2_subscribers(&mut self, chain_id: &ChainId) -> Result<(), Error> {
      self.directives
          .iter()
          .map(|(_, d)| d.clone())
          .collect::<Vec<Directive>>()
          .into_iter()
          .for_each(|d| {
              info!(
                  "republish directives({:?}) for subscribers: {}",
                  d,
                  chain_id.to_string()
              );
              let _ = self.pub_2_subscribers(Some(vec![chain_id.clone()]), d);
          });
```

```


    Ok(())
}
```

Listing 2.24: hub/src/state.rs

Suggestion Remove the redundant function `repub_2_subscribers()`.

2.4 Note

2.4.1 Potential centralized risks

Introduced by `Version 1`

Description The canisters' controller or admin can upgrade canisters and execute critical tasks (e.g., `send_ticket()`) in the hub, which may bring centralized risks. Thus, the privileged accounts' private keys should be kept safe.

2.4.2 Tickets are processed in the txid order

Introduced by `Version 1`

Description The `get_pending_gen_ticket_requests()` will return a bunch of tickets sorted by their `txid`. Since `txid` is random for users, a ticket submitted earlier may be processed later.

2.4.3 Lack of cross-chain capability for multiple rune types or destinations in one Bitcoin transaction

Introduced by `Version 1`

Description Function `generate_ticket()` can only generate a single request for one user-provided rune type and one destination address, which means each transaction is limited to a single rune type and destination. Once generated, the `txid` is recorded as a key in the `pending_gen_ticket_requests`, preventing its reuse. As a result, if users transfer multiple types of runes or transfer to multiple destinations in one transaction, only one of the rune types and the destinations can be executed successfully.

```
pub async fn generate_ticket(args: GenerateTicketArgs) -> Result<(), GenerateTicketError> {
    if read_state(|s| s.chain_state == ChainState::Deactive) {
        return Err(GenerateTicketError::TemporarilyUnavailable(
            "chain state is deactive!".into(),
        ));
    }


    init_ecdsa_public_key().await;
    let _guard = generate_ticket_guard()?;


    let rune_id = RuneId::from_str(&args.rune_id)
```

```
            .map_err(|e| GenerateTicketError::InvalidRuneId(e.to_string()))?;


        let txid = Txid::from_str(&args.txid).map_err(|_| GenerateTicketError::InvalidTxId)?;


        if !read_state(|s| {
            s.counterparties
                .get(&args.target_chain_id)
                .is_some_and(|c| c.chain_state == ChainState::Active)
        }) {
            return Err(GenerateTicketError::UnsupportedChainId(
                args.target_chain_id.clone(),
            ));
        }


        let token_id = read_state(|s| {
            if let Some((token_id, _)) = s.tokens.iter().find(|(_, (r, _))| rune_id.eq(r)) {
                Ok(token_id.clone())
            } else {
                Err(GenerateTicketError::UnsupportedToken(args.rune_id))
            }
        })?;


        read_state(|s| match s.generate_ticket_status(txid) {
            GenTicketStatus::Pending(_) => Err(GenerateTicketError::AlreadySubmitted),
            GenTicketStatus::Invalid | GenTicketStatus::Finalized => {
                Err(GenerateTicketError::AleardyProcessed)
            }
            GenTicketStatus::Unknown => Ok(()),
        })?;


        let (btc_network, min_confirmations) = read_state(|s| (s.btc_network, s.min_confirmations))
            ;


        let destination = Destination {
            target_chain_id: args.target_chain_id.clone(),
            receiver: args.receiver.clone(),
            token: None,
        };


        let address = read_state(|s| destination_to_p2wpkh_address_from_state(s, &destination));


        // In order to prevent the memory from being exhausted,
        // ensure that the user has transferred token to this address.
        let utxos = get_utxos(btc_network, &address, min_confirmations, CallSource::Client)
            .await
            .map_err(|call_err| {
                GenerateTicketError::TemporarilyUnavailable(format!(
                    "Failed to call bitcoin canister: {}",
                    call_err
                ))
            })?
            .utxos;


        let new_utxos = read_state(|s| s.new_utxos(utxos, Some(txid)));
        if new_utxos.len() == 0 {
            return Err(GenerateTicketError::NoNewUtxos);
        }


        let request = GenTicketRequest {
            address,
            target_chain_id: args.target_chain_id,
            receiver: args.receiver,
            token_id,
            rune_id,
            amount: args.amount,
            txid,
            received_at: ic_cdk::api::time(),
        };


        mutate_state(|s| {
            audit::accept_generate_ticket_request(s, request);
            audit::add_utxos(s, destination, new_utxos, true);
        });
        Ok(())
    }
```

Listing 2.25: customs/bitcoin/src/updates/generate_ticket.rs

Feedback from the project This is by design. There is only one rune token across chains at a time.

2.4.4 Potential temporary block of cross-chain requests due to deactivation of chains

Introduced by Version 1

Description According to the design, privileged accounts have the authority to execute approved proposals in the hub canister to deactivate chains. After deactivation, the corresponding Bitcoin and Route canisters will synchronize related information after the proposal execution by calling process_directives(). However, in the Route canister, process_directives() is called before process_tickets() which handles related cross-chain requests. If at this time the chain state has already changed to deactivated, all requests will fail. Similar failures would also occur in the following situations:

- Deactivation occurs after the user has already transferred runes to the corresponding `Bitcoin` addresses but before calling `generate_ticket()`.
- Deactivation occurs after the user successfully generates the ticket but before the oracle calls `update_runes_balance()`.
- Deactivation occurs before the function `send_tickets()` is executed in the `Route` and `Bitcoin` canisters.

```rust
pub async fn execute_proposal(proposals: Vec<Proposal>) -> Result<(), Error> {
    for proposal in proposals.into_iter() {
        match proposal {
            Proposal::AddChain(chain_meta) => {
                // save new chain
                with_state_mut(|hub_state| {
                    info!(" save new chain: {:?}", chain_meta);
                    hub_state.add_chain(chain_meta.clone())
                })?;
                // publish directive for the new chain)
                info!(
                    "publish directive for ‘AddChain‘ proposal :{:?}",
                    chain_meta.to_string()
                );
                with_state_mut(|hub_state| {
                    let target_subs = chain_meta.counterparties.clone().unwrap_or_default();
                    hub_state
                        .pub_directive(Some(target_subs), &Directive::AddChain(chain_meta.into())
                            )
                })?;
            }


            Proposal::AddToken(token_meata) => {
                info!(
                    "publish directive for ‘AddToken‘ proposal :{:?}",
                    token_meata
                );


                with_state_mut(|hub_state| {
                    // save token info
                    hub_state.add_token(token_meata.clone())?;
                    // publish directive
                    hub_state.pub_directive(
                        Some(token_meata.dst_chains.clone()),
                        &Directive::AddToken(token_meata.into()),
                    )
                })?
            }


            Proposal::ToggleChainState(toggle_status) => {
                info!(
                    "publish directive for ‘ToggleChainState‘ proposal :{:?}",
                    toggle_status
                );


                with_state_mut(|hub_state| {
                    // publish directive
                    hub_state
                        .pub_directive(None, &Directive::ToggleChainState(toggle_status.clone()))
                            ?;
                    // update dst chain state
                    hub_state.update_chain_state(&toggle_status)
                })?;
            }


            Proposal::UpdateFee(factor) => {
                info!("publish directive for ‘UpdateFee‘ proposal :{:?}", factor);
                with_state_mut(|hub_state| {
                    hub_state.update_fee(factor.clone())?;
                    let target_subs = match &factor {
                        Factor::UpdateTargetChainFactor(factor) => {
                            hub_state.get_chains_by_counterparty(factor.target_chain_id.clone())
                        }
                        Factor::UpdateFeeTokenFactor(factor) => {
                            hub_state.get_chains_by_fee_token(factor.fee_token.clone())
                        }
                    };
                    hub_state
                        .pub_directive(Some(target_subs), &Directive::UpdateFee(factor.clone()))
                })?;
            }
        }
    }
    Ok(())
}
```

Listing 2.26: hub/src/proposal.rs

```
async fn process_directives() {
    let (hub_principal, offset) = read_state(|s| (s.hub_principal, s.next_directive_seq));
    match hub::query_directives(hub_principal, offset, BATCH_QUERY_LIMIT).await {
        Ok(directives) => {
            for (_, directive) in &directives {
                match directive {
                    Directive::AddChain(chain) => {
                        mutate_state(|s| audit::add_chain(s, chain.clone()));
                    }
                    Directive::AddToken(token) => {
                        match updates::add_new_token(token.clone()).await {
                            Ok(_) => {
                                log::info!(
                                    "[process directives] add token successful, token id: {}",
                                    token.token_id
                                );
                            }
                            Err(err) => {
                                log::error!(
                                    "[process directives] failed to add token: token id: {}, err: {:?}",
                                    token.token_id,
                                    err
                                );
                            }
                        }
                    }
                    Directive::ToggleChainState(toggle) => {
                        mutate_state(|s| audit::toggle_chain_state(s, toggle.clone()));
                    }
                    Directive::UpdateFee(fee) => {
                        mutate_state(|s| audit::update_fee(s, fee.clone()));
                        log::info!("[process_directives] success to update fee, fee: {}", fee);
                    }
                }
            }
            let next_seq = directives.last().map_or(offset, |(seq, _)| seq + 1);
            mutate_state(|s| {
                s.next_directive_seq = next_seq;
            });
        }
        Err(err) => {
            log::error!(
                "[process directives] failed to query directives, err: {:?}",
                err
            );
        }
    };
}


pub fn periodic_task() {
    ic_cdk::spawn(async {
        let _guard = match crate::guard::TimerLogicGuard::new() {
            Some(guard) => guard,
            None => return,
        };


        process_directives().await;
        process_tickets().await;
    });
}
```

Listing 2.27: route/icp/lib.rs

Feedback from the project Generally, the deactive chain only occurs when there is a problem and needs to be updated. After the update is completed, the chain will be reactivated, so that tickets that have not been processed before will continue to be processed.

2.4.5 Lack of refunding mechanism for user's mistaken operations

Introduced by Version 1

Description The current implementation does not support refunds for users' mistaken cross-chain operations. Specifically, if a user wants to transfer runes from Bitcoin to a destination chain, the user should transfer runes to the specified Bitcoin address first. However, if the destination chain is not supported currently, invoking generate_ticket() will revert due to the target chain's state being checked as Deactive. In this case, the protocol does not have a relevant refunding mechanism to return the runes transferred by the user.

A similar failure also exists in the function update_runes_balance(). It verifies requests composed of user-provided parameters and reverts if they are not matched with parameters provided by the oracle. It is possible for users to successfully transfer runes to the specified Bitcoin address but fill in the wrong parameters (e.g., runes amount) when generating the requests. Runes will not be refunded in this case either.

```rust
pub async fn generate_ticket(args: GenerateTicketArgs) -> Result<(), GenerateTicketError> {
    if read_state(|s| s.chain_state == ChainState::Deactive) {
        return Err(GenerateTicketError::TemporarilyUnavailable(
            "chain state is deactive!".into(),
        ));
    }


    init_ecdsa_public_key().await;
    let _guard = generate_ticket_guard()?;


    let rune_id = RuneId::from_str(&args.rune_id)
        .map_err(|e| GenerateTicketError::InvalidRuneId(e.to_string()))?;


    let txid = Txid::from_str(&args.txid).map_err(|_| GenerateTicketError::InvalidTxId)?;


    if !read_state(|s| {
        s.counterparties
            .get(&args.target_chain_id)
            .is_some_and(|c| c.chain_state == ChainState::Active)
    }) {
        return Err(GenerateTicketError::UnsupportedChainId(
            args.target_chain_id.clone(),
        ));
    }


    let token_id = read_state(|s| {
        if let Some((token_id, _)) = s.tokens.iter().find(|(_, (r, _))| rune_id.eq(r)) {
            Ok(token_id.clone())
        } else {
            Err(GenerateTicketError::UnsupportedToken(args.rune_id))
        }
    })?;


    read_state(|s| match s.generate_ticket_status(txid) {
        GenTicketStatus::Pending(_) => Err(GenerateTicketError::AlreadySubmitted),
        GenTicketStatus::Invalid | GenTicketStatus::Finalized => {
            Err(GenerateTicketError::AleardyProcessed)
        }
        GenTicketStatus::Unknown => Ok(()),
    })?;


    let (btc_network, min_confirmations) = read_state(|s| (s.btc_network, s.min_confirmations))
        ;


    let destination = Destination {
        target_chain_id: args.target_chain_id.clone(),
        receiver: args.receiver.clone(),
        token: None,
    };


    let address = read_state(|s| destination_to_p2wpkh_address_from_state(s, &destination));


    // In order to prevent the memory from being exhausted,
    // ensure that the user has transferred token to this address.
    let utxos = get_utxos(btc_network, &address, min_confirmations, CallSource::Client)
        .await
        .map_err(|call_err| {
            GenerateTicketError::TemporarilyUnavailable(format!(
                "Failed to call bitcoin canister: {}",
                call_err
            ))
        })?
        .utxos;


    let new_utxos = read_state(|s| s.new_utxos(utxos, Some(txid)));
    if new_utxos.len() == 0 {
        return Err(GenerateTicketError::NoNewUtxos);
    }


    let request = GenTicketRequest {
        address,
        target_chain_id: args.target_chain_id,
        receiver: args.receiver,
        token_id,
        rune_id,
        amount: args.amount,
        txid,
        received_at: ic_cdk::api::time(),
    };


    mutate_state(|s| {
        audit::accept_generate_ticket_request(s, request);
        audit::add_utxos(s, destination, new_utxos, true);
    });
    Ok(())
}
```

Listing 2.28: customs/bitcoin/src/updates/generate_ticket.rs

```
pub async fn update_runes_balance(
    args: UpdateRunesBalanceArgs,
) -> Result<(), UpdateRunesBalanceError> {
    for balance in &args.balances {
        let outpoint = OutPoint {
            txid: args.txid,
            vout: balance.vout,
        };
        read_state(|s| match s.outpoint_destination.get(&outpoint) {
            Some(_) => Ok(()),
            None => Err(UpdateRunesBalanceError::UtxoNotFound),
        })?;
    }


    let req = read_state(|s| match s.generate_ticket_status(args.txid) {
        GenTicketStatus::Invalid | GenTicketStatus::Finalized => {
            Err(UpdateRunesBalanceError::AleardyProcessed)
        }
        GenTicketStatus::Unknown => Err(UpdateRunesBalanceError::RequestNotFound),
        GenTicketStatus::Pending(req) => Ok(req),
    })?;


    let amount = args.balances.iter().map(|b| b.amount).sum::<u128>();
    if amount != req.amount || args.balances.iter().any(|b| b.rune_id != req.rune_id) {
        return Err(UpdateRunesBalanceError::MismatchWithGenTicketReq);
    }


    let (hub_principal, chain_id) = read_state(|s| (s.hub_principal, s.chain_id.clone()));
    hub::send_ticket(
        hub_principal,
        Ticket {
            ticket_id: args.txid.to_string(),
            ticket_type: TicketType::Normal,
            ticket_time: ic_cdk::api::time(),
            src_chain: chain_id,
            dst_chain: req.target_chain_id.clone(),
            action: TxAction::Transfer,
            token: req.token_id.clone(),
            amount: req.amount.to_string(),
            sender: None,
            receiver: req.receiver.clone(),
            memo: None,
        },
    )
    .await
    .map_err(|err| UpdateRunesBalanceError::SendTicketErr(format!("{}", err)))?;


    mutate_state(|s| audit::finalize_ticket_request(s, &req, args.balances));


    Ok(())
}
```

Listing 2.29: customs/bitcoin/src/updates/update_runes_balance.rs

Feedback from the project The cost of refunding on the chain is relatively high. If the chain is temporarily deactivated, the user can wait for activation before calling `generate_ticket()` (this time is generally not too long). If it is for other reasons, we usually handle it manually. As long as the user correctly follows the front-end instructions, abnormal situations are unlikely to occur.

2.4.6 Inconsistency of cross-chain runes amount limitation

Introduced by `Version 1`

Description The protocol does not limit the minimum amount of runes cross-chained from the `Bitcoin` chain to the target chain, but it limits the number of runes redeemed back to `Bitcoin` from the target chain (i.e., `min_burn_amount`). This implementation is inconsistent and may likely cause the runes crossed to the target chain by the user to be unable to be redeemed, forcing the user to cross another sufficient amount of runes to the target chain to meet the minimum value check in order to complete the redemption operation.

```
pub async fn generate_ticket(
    req: GenerateTicketReq,
) -> Result<GenerateTicketOk, GenerateTicketError> {
    if read_state(|s| s.chain_state == ChainState::Deactive) {
        return Err(GenerateTicketError::TemporarilyUnavailable(
            "chain state is deactive!".into(),
        ));
    }


    if !read_state(|s| {
        s.counterparties
            .get(&req.target_chain_id)
            .is_some_and(|c| c.chain_state == ChainState::Active)
    }) {
        return Err(GenerateTicketError::UnsupportedChainId(
            req.target_chain_id.clone(),
        ));
    }


    let ledger_id = read_state(|s| match s.token_ledgers.get(&req.token_id) {
        Some(ledger_id) => Ok(ledger_id.clone()),
        None => Err(GenerateTicketError::UnsupportedToken(req.token_id.clone())),
    })?;


    charge_redeem_fee(caller(), &req.target_chain_id).await?;


    let caller = ic_cdk::caller();
    let user = Account {
        owner: caller,
        subaccount: req.from_subaccount,
    };


    let block_index = burn_token_icrc2(ledger_id, user, req.amount).await?;
    let ticket_id = format!("{}_{}", ledger_id.to_string(), block_index.to_string());


    let (hub_principal, chain_id) = read_state(|s| (s.hub_principal, s.chain_id.clone()));
    hub::send_ticket(
        hub_principal,
        Ticket {
            ticket_id: ticket_id.clone(),
            ticket_type: omnity_types::TicketType::Normal,
            ticket_time: ic_cdk::api::time(),
            src_chain: chain_id,
            dst_chain: req.target_chain_id.clone(),
            action: TxAction::Redeem,
            token: req.token_id.clone(),
            amount: req.amount.to_string(),
            sender: None,
            receiver: req.receiver.clone(),
            memo: None,
        },
    )
    .await
    .map_err(|err| GenerateTicketError::SendTicketErr(format!("{}", err)))?;


    audit::finalize_gen_ticket(ticket_id.clone(), req);
    Ok(GenerateTicketOk { ticket_id })
}

```

```

async fn burn_token_icrc2(
    ledger_id: Principal,
    user: Account,
    amount: u128,
) -> Result<u64, GenerateTicketError> {
    let client = ICRC1Client {
        runtime: CdkRuntime,
        ledger_canister_id: ledger_id,
    };
    let route = ic_cdk::id();
    let result = client
        .transfer_from(TransferFromArgs {
            spender_subaccount: None,
            from: user,
            to: Account {
                owner: route,
                subaccount: None,
            },
            amount: Nat::from(amount),
            fee: None,
            memo: None,
            created_at_time: Some(ic_cdk::api::time()),
        })
        .await
        .map_err(|(code, msg)| {
            GenerateTicketError::TemporarilyUnavailable(format!(
                "cannot enqueue a burn transaction: {} (reject_code = {})",
                msg, code
            ))
        })?;


    match result {
        Ok(block_index) => Ok(block_index.0.to_u64().expect("nat does not fit into u64")),
        Err(TransferFromError::InsufficientFunds { balance }) => Err(GenerateTicketError::
             InsufficientFunds {
            balance: balance.0.to_u64().expect("unreachable: ledger balance does not fit into
                 u64")
        }),
        Err(TransferFromError::InsufficientAllowance { allowance }) => Err(GenerateTicketError
             ::InsufficientAllowance {
            allowance: allowance.0.to_u64().expect("unreachable: ledger balance does not fit
                 into u64")
        }),
        Err(TransferFromError::TemporarilyUnavailable) => {
            Err(GenerateTicketError::TemporarilyUnavailable(
                "cannot burn token: the ledger is busy".to_string(),
            ))
        }
        Err(TransferFromError::GenericError { error_code, message }) => {
            Err(GenerateTicketError::TemporarilyUnavailable(format!(
                "cannot burn token: the ledger fails with: {} (error code {})", message,
                    error_code
            )))
        }
        Err(TransferFromError::BadFee { expected_fee }) => ic_cdk::trap(&format!(
            "unreachable: the ledger demands the fee of {} even though the fee field is unset",
            expected_fee
        )),
        Err(TransferFromError::Duplicate { duplicate_of }) => ic_cdk::trap(&format!(
            "unreachable: the ledger reports duplicate ({}) even though the create_at_time
                 field is unset",
            duplicate_of
        )),
        Err(TransferFromError::CreatedInFuture {..}) => ic_cdk::trap(
            "unreachable: the ledger reports CreatedInFuture even though the create_at_time
                 field is unset"
        ),
        Err(TransferFromError::TooOld) => ic_cdk::trap(
            "unreachable: the ledger reports TooOld even though the create_at_time field is
                 unset"
        ),
        Err(TransferFromError::BadBurn { min_burn_amount }) => ic_cdk::trap(&format!(
            "the burn amount {} is less than ledger's min_burn_amount {}",
            amount,
            min_burn_amount
        )),
    }
}
```

Listing 2.30: route/icp/src/updates/generate_ticket.rs

Feedback from the project `min_burn_amount` is defined by `icrc1` ledger, which means that the amount of burn must be greater than `transfer_fee` (this amount is very small). Assuming that the token the user crosses is smaller than the transfer fee, there is actually no need to redeem it.

2.4.7 Potential insufficient fees for Bitcoin resubmissions

Introduced by `Version 1`

Description In function `build_unsigned_transaction()`, an unsigned transaction will be built if there are enough `Bitcoin` inputs to cover the fees. However, the final check doesn't account for the resubmission fees, which can lead to resubmission failure. As shown in the code, when `input_btc_amount` equals to `btc_consumed` (if number of BTC inputs is further larger than 2), resubmission will always fail as the `fee_per_vbyte` of resubmission is larger than the original one.

```
    // We need to recaculate the fee when the number of inputs and outputs is finalized.
    let real_fee = fake_sign(&unsigned_tx).vsize() as u64 * fee_per_vbyte / 1000;
    let btc_consumed = real_fee + MIN_OUTPUT_AMOUNT * non_op_return_outputs_sz;
    if input_btc_amount < btc_consumed {
        log!(
            P0,
            "input btc amount: {} greater than btc consumed: {}",
            input_btc_amount,
            btc_consumed,
        );
        return Err(BuildTxError::NotEnoughGas);
    }


    let btc_change_amount = input_btc_amount - btc_consumed + MIN_OUTPUT_AMOUNT;
    unsigned_tx.outputs.iter_mut().last().unwrap().value = btc_change_amount;
    let btc_change_out = BtcChangeOutput {
        vout: unsigned_tx.outputs.len() as u32 - 1,
        value: btc_change_amount,
    };


    Ok((
        unsigned_tx,
        change_output,
        btc_change_out,
        ScopeGuard::into_inner(runes_utxos_guard),
        ScopeGuard::into_inner(btc_utxos_guard),
    ))
```

Listing 2.31: customs/bitcoin/src/lib.rs

Feedback from the project The amount of BTC selected is twice the current fee. Normally it will not be insufficient. Even if it is insufficient due to repeated resubmit, we can wait for the BTC network fee to drop before the miners package the transaction.

2.4.8 Potential insufficient cycles in upgrade

Introduced by `Version 1`

Description The `Bitcoin` custom will replay all the past events to change the canister's state. If there are enormous events, the `post_upgrade()` will fail because of insufficient cycles.

```
pub fn replay(mut events: impl Iterator<Item = Event>) -> Result<CustomsState, ReplayLogError>
    {
    let mut state = match events.next() {
        Some(Event::Init(args)) => CustomsState::from(args),
        Some(evt) => {
            return Err(ReplayLogError::InconsistentLog(format!(
                "The first event is not Init: {:?}",
                evt
            )))
        }
        None => return Err(ReplayLogError::EmptyLog),
    };


    for event in events {
        match event {
```

```rust
                Event::Init(args) => {
                    state.reinit(args);
                }
                Event::Upgrade(args) => state.upgrade(args),
                Event::AddedChain(chain) => {
                    state.counterparties.insert(chain.chain_id.clone(), chain);
                }
                Event::AddedToken { rune_id, token } => {
                    state
                        .tokens
                        .insert(token.token_id.clone(), (rune_id, token));
                }
                Event::ToggleChainState(toggle) => {
                    if toggle.chain_id == state.chain_id {
                        state.chain_state = toggle.action.into();
                    } else if let Some(chain) = state.counterparties.get_mut(&toggle.chain_id) {
                        chain.chain_state = toggle.action.into();
                    }
                }
                Event::UpdateNextDirectiveSeq(next_seq) => {
                    assert!(next_seq > state.next_directive_seq);
                    state.next_directive_seq = next_seq;
                }
                Event::UpdateNextTicketSeq(next_seq) => {
                    assert!(next_seq > state.next_ticket_seq);
                    state.next_ticket_seq = next_seq;
                }
                Event::ReceivedUtxos {
                    destination,
                    utxos,
                    is_runes,
                } => state.add_utxos(destination, utxos, is_runes),
                Event::UpdatedRunesBalance { txid, balance } => {
                    state.update_runes_balance(txid, balance);
                }
                Event::AcceptedGenTicketRequest(req) => {
                    state.pending_gen_ticket_requests.insert(req.txid, req);
                }
                Event::FinalizedTicketRequest { txid, balances } => {
                    let request = state
                        .pending_gen_ticket_requests
                        .remove(&txid)
                        .ok_or_else(|| {
                            ReplayLogError::InconsistentLog(format!(
                                "Attempted to remove a non-pending generate ticket request {}",
                                txid
                            ))
                        })?;
                    for balance in balances {
                        state.update_runes_balance(txid, balance);
                    }
                    state.push_finalized_ticket(request);
                }
```

```
            Event::AcceptedReleaseTokenRequest(req) => {
                state.push_back_pending_request(req);
            }
            Event::SentBtcTransaction {
                rune_id,
                request_release_ids,
                txid,
                runes_utxos,
                btc_utxos,
                fee_per_vbyte,
                runes_change_output,
                btc_change_output,
                submitted_at,
            } => {
                let mut release_token_requests = Vec::with_capacity(request_release_ids.len());
                for release_id in request_release_ids {
                    let request = state
                        .remove_pending_request(release_id.clone())
                        .ok_or_else(|| {
                            ReplayLogError::InconsistentLog(format!(
                                "Attempted to send a non-pending release_token request {:?}",
                                release_id
                            ))
                        })?;
                    release_token_requests.push(request);
                }
                for utxo in runes_utxos.iter() {
                    state.available_runes_utxos.remove(utxo);
                }
                for utxo in btc_utxos.iter() {
                    state.available_fee_utxos.remove(utxo);
                }
                state.push_submitted_transaction(SubmittedBtcTransaction {
                    rune_id,
                    requests: release_token_requests,
                    txid,
                    runes_utxos,
                    btc_utxos,
                    fee_per_vbyte,
                    runes_change_output,
                    btc_change_output,
                    submitted_at,
                });
            }
            Event::ReplacedBtcTransaction {
                old_txid,
                new_txid,
                runes_change_output,
                btc_change_output,
                submitted_at,
                fee_per_vbyte,
            } => {
                let (requests, runes_utxos, btc_utxos) = match state
```

```
                        .submitted_transactions
                        .iter()
                        .find(|tx| tx.txid == old_txid)
                    {
                        Some(tx) => (
                            tx.requests.clone(),
                            tx.runes_utxos.clone(),
                            tx.btc_utxos.clone(),
                        ),
                        None => {
                            return Err(ReplayLogError::InconsistentLog(format!(
                                "Cannot replace a non-existent transaction {}",
                                &old_txid
                            )))
                        }
                    };


                    state.replace_transaction(
                        &old_txid,
                        SubmittedBtcTransaction {
                            rune_id: runes_change_output.rune_id.clone(),
                            txid: new_txid,
                            requests,
                            runes_utxos,
                            btc_utxos,
                            runes_change_output,
                            btc_change_output,
                            submitted_at,
                            fee_per_vbyte: Some(fee_per_vbyte),
                        },
                    );
                }
                Event::ConfirmedBtcTransaction { txid } => {
                    state.finalize_transaction(&txid);
                }
            }
        }


        Ok(state)
    }
```

Listing 2.32: customs/bitcoin/src/state/eventlog.rs

Feedback from the project The upgrade method of replay event follows `ckbtc`. The advantage is that adding fields to the state does not affect the upgrade. `ckbtc` events should be more than ours, and the cycle consumed does not seem to be much at present.

2.4.9 Potential double spending by resubmitted tickets

Introduced by `Version 1`

Description In the hub canister, privileged accounts have the authority to resubmit already submitted tickets. However, the old tickets will not be replaced and can still be queried and proceeded by the corresponding target chain's canister. Since the target chain's canister currently does not perform any validation on the queried tickets, this allows both the original and resubmitted ticket corresponding to the same cross-chain request to be processed normally.

```rust
 pub fn push_ticket(&mut self, ticket: Ticket) -> Result<(), Error> {
    // get latest ticket seq
    let latest_ticket_seq = self
        .ticket_seq
        .entry(ticket.dst_chain.to_string())
        .and_modify(|seq| *seq += 1)
        .or_insert(0);


    // add new ticket
    let seq_key = SeqKey::from(ticket.dst_chain.to_string(), *latest_ticket_seq);
    self.ticket_queue.insert(seq_key.clone(), ticket.clone());
    //save ticket
    self.cross_ledger
        .insert(ticket.ticket_id.to_string(), ticket.clone());
    record_event(&Event::ReceivedTicket {
        seq_key,
        ticket: ticket.clone(),
    });
    Ok(())
  }


  pub fn resubmit_ticket(&mut self, ticket: Ticket) -> Result<(), Error> {
    let now = ic_cdk::api::time();
    if now - self.last_resubmit_ticket_time < 6 * HOUR {
        return Err(Error::ResubmitTicketSentTooOften);
    }
    match self.cross_ledger.get(&ticket.ticket_id) {
        Some(old_ticket) => {
            if ticket != old_ticket {
                return Err(Error::ResubmitTicketMustSame);
            }
            let ticket_id = format!("{}_{}", ticket.ticket_id, now);
            let new_ticket = Ticket {
                ticket_id: ticket_id.clone(),
                ticket_type: TicketType::Resubmit,
                ticket_time: now,
                src_chain: ticket.src_chain,
                dst_chain: ticket.dst_chain,
                action: ticket.action,
                token: ticket.token,
                amount: ticket.amount,
                sender: ticket.sender,
                receiver: ticket.receiver,
                memo: ticket.memo,
            };
            self.push_ticket(new_ticket)?;
            self.last_resubmit_ticket_time = now;


            record_event(&Event::ResubmitTicket {
                ticket_id,
                timestamp: now,
            });
            Ok(())
        }
        None => Err(Error::ResubmitTicketIdMustExist),
    }
}
```

Listing 2.33: hub/src/state.rs

Feedback from the project `resubmit_ticket` is used in special emergency situations when it is first launched. Only when it is determined that the original ticket must not be processed successfully will the interface be manually called. The interface should be offline after the system is stable.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS