BLOCKSEC

Security Audit Report for Side Protocol

Date: January 7, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Side Protocol
Target	Side Protocol

Version History

Version	Date	Description
1.0	January 7, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-01-13 02:13:22+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	Cosmos Chain & Software Implementation
Language	Rust & Go
Approach	Semi-automatic and manual verification

This audit focuses on the code repositories of Shuttler [1], FROST [2] and Side Chain [3] of Side Protocol. Side Protocol, as an extension layer of Bitcoin, is the first fully Bitcoin-compatible dPoS Layer 1 blockchain, designed to shape the future of Bitcoin finance. It enables developers to create secure, high-performance decentralized applications within the Bitcoin-centric internet, aiming to onboard billions of users globally and establish BTC as the definitive global currency.

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
Shuttler	Version 1	3b372bf9ea005272302fe4f6ca79e0cd50071408
	Version 2	f8498b0049906e44d2a37b89a1661c793014d7ac
FROST	Version 1	ab3251bf47c43c7a4d3e286c7ba8129306aa1e3e
	Version 2	3ed3d179c240b07c022544edbb561839366bb4aa
Side Chain	Version 1	a26318e6b6dd6fc315c0d63a32ffc10caa390f85
	Version 2	43dfc21c4470a543a3efb6f2246ff13cd319e2c2

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/sideprotocol/shuttler

[2] https://github.com/sideprotocol/frost

[3] https://github.com/sideprotocol/side

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 and Common Weakness Enumeration. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	High	Low
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

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

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **eight** potential security issues. Besides, we have **three** recommendations and **seven** notes.

- High Risk: 3
- Medium Risk: 1
- Low Risk: 4
- Recommendation: 3
- Note: 7

ID	Severity	Description	Category	Status
1	Medium	Improper sender address validation in function `received_dkg_response()`	Software Security	Fixed
2	Low	Potential invalid vault address submission to side chain due to failure to generate round 2 packages	Software Security	Fixed
3	Low	Lack of task existence check in function `received_sign_message()`	Software Security	Confirmed
4	High	Insufficient check on block headers	Software Security	Fixed
5	High	Potential nonce reuse during signature generation process of Round 1	Software Security	Fixed
6	Low	Lack of duplication checks for participants during initiating DKG	Software Security	Fixed
7	High	DoS due to inconsistent vault versions	Software Security	Fixed
8	Low	Lack of error handling logic in function `scan_vault_txs()`	Software Security	Fixed
9	-	Lack of error handling logic in function `submit_signature()`	Recommendation	Confirmed
10	-	Fix potential panics	Recommendation	Fixed
11	-	Remove unused code	Recommendation	Fixed
12	-	Potential centralization risk	Note	-
13	-	Trusted participants in the DKG process	Note	-
14	-	Frozen protocol fees	Note	-
15	-	Rune verification reliance on external mechanisms	Note	-
16	-	Potential DoS due to expired fee rate	Note	-
17	-	Potential DoS due to excessive unrelated commitments in Round 1 message	Note	-
18	-	Private keys and passwords are stored in plaintext format	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Improper sender address validation in function `received_dkg_response()`

Severity Medium

Status Fixed in `Version 2` (Shuttler)

Introduced by `Version 1` (Shuttler)

Description The `received_dkg_response()` function attempts to verify the `sender` of a DKG response by calculating the sender's address using a SHA256 hash and checking whether it exists in the task's participant list. However, this validation mechanism can be circumvented because the sender's address can be easily spoofed. Specifically, an attacker can fetch the DKG request and impersonate any listed participant. Consequently, attackers are capable of creating a `DKGResponse` that appears valid and passes the participant verification check. Furthermore, attackers can inject arbitrary `round1_packages` or `round2_packages` into the local field, potentially inflating its size beyond the number of legitimate participants. This manipulation could disrupt the protocol, as the code relies on the length of `local` to determine whether all participants have submitted their packages. An inflated `local` length will prevent subsequent DKG process.

```rust
pub fn received_dkg_response(response: DKGResponse, signer: &Signer) {
    let task_id = response.payload.task_id.clone();
    let mut task = match signer.get_dkg_task(&task_id) {
        Some(task) => task,
        None => {
            return;
        }
    };

    let addr = sha256::digest(&response.sender.serialize())[0..40].to_uppercase();
    if !task.participants.contains(&addr) {
        debug!("Invalid DKG participant {:?}, {:?}", response.sender, addr);
        return;
    }

    if task.round == Round::Round1 {
        received_round1_packages(&mut task, response.payload.round1_packages, signer)
    } else if task.round == Round::Round2 {
        received_round2_packages(&mut task, response.payload.round2_packages, signer)
    }
}
```

Listing 2.1: src/apps/signer/dkg.rs

```rust
pub fn received_round1_packages(task: &mut DKGTask, packets: BTreeMap<Identifier, keys::dkg::
    round1::Package>, signer: &Signer) {

// store round 1 packets
```

```
let mut local = signer.get_dkg_round1_package(&task.id).map_or(BTreeMap::new(), |v|v);

// merge packets with local
local.extend(packets);
signer.save_dkg_round1_package(&task.id, &local);

// let k = local.keys().map(|k| to_base64(&k.serialize()[..])).collect::<Vec<_>>();
debug!("Received round1 packets: {} {:?}", task.id, local.keys());

// if DB.insert(format!("dkg-{}-round1", task.id), serde_json::to_vec(&local).unwrap()).is_err
    () {
//     error!("Failed to store DKG Round 1 packets: {} ", task.id);
// }

if task.participants.len() == local.len() {

    info!("Received round1 packets from all participants: {}", task.id);
    match generate_round2_packages(&signer, task, local) {
        Ok(_) => {
            task.round = Round::Round2;
            signer.save_dkg_task(&task);
        }
        Err(e) => {
            task.round = Round::Closed;
            signer.save_dkg_task(&task);
            error!("Failed to generate round2 packages: {} - {:?}", task.id, e);
        }
    }
    return;
}
}
```

Listing 2.2: src/apps/signer/dkg.rs

Impact The DKG process may fail to complete successfully.

Suggestion Revise the logic in the `received_dkg_response()` function to include verification of the `DKGResponse signature`.

2.1.2 Potential invalid vault address submission to side chain due to failure to generate round 2 packages

Severity Low

Status Fixed in `Version 2` (Shuttler)

Introduced by `Version 1` (Shuttler)

Description When the `frost::keys::dkg::part2()` function fails, the subsequent action by the `generate_round2_packages()` function will return `Err(DKGError())`. Following this error, the `received_round1_packages()` function updates the task status to `CLOSE`. Despite this state indicating a failure, the `submit_dkg_address()` function continues to process tasks marked as `CLOSE`. Specifically, it attempts to submit the vault address to the Side Chain even if these tasks have not been successfully completed. This submission is ineffective under a failure condition and can lead to further complications, as the Side Chain is provided with an invalid or non-existent address.

```rust
pub fn generate_round2_packages(signer: &Signer, task: &mut DKGTask, round1_packages: BTreeMap<Identifier, Package<Secp256K1Sha256>>) -> Result<(), DKGError> {

    let task_id = task.id.clone();

    let secret_package = match mem_store::get_dkg_round1_secret_packet(&task_id) {
        Some(secret_packet) => secret_packet,
        None => {
            return Err(DKGError(format!("No secret packet found for DKG: {}", task_id)));
        }
    };

    if task.participants.len() as u16 != round1_packages.len() as u16 {
        return Err(DKGError(format!("Have not received enough packages: {}", task_id)));
    }

    let mut cloned = round1_packages.clone();
    cloned.remove(signer.identifier());

    match frost::keys::dkg::part2(secret_package, &cloned) {
        Ok((round2_secret_package, round2_packages)) => {
            mem_store::set_dkg_round2_secret_packet(&task_id, round2_secret_package);

            // convert it to <receiver, Vec<u8>>, then only the receiver can decrypt it.
            let mut output_packages = BTreeMap::new();
            for (receiver_identifier, round2_package) in round2_packages {
                let bz = receiver_identifier.serialize();
                let target = x25519::PublicKey::from_ed25519(&ed25519_compact::PublicKey::from_slice(bz.as_slice()).unwrap()).unwrap();

                let share_key = target.dh(&x25519::SecretKey::from_ed25519(&signer.identity_key).unwrap()).unwrap();

                let byte = round2_package.serialize().unwrap();
                let packet = encrypt(byte.as_slice(), share_key.as_slice().try_into().unwrap());

                output_packages.insert(receiver_identifier, packet);
            };

            // convert it to <sender, <receiver, Vec<u8>>
            let mut merged = BTreeMap::new();
            merged.insert(signer.identifier().clone(), output_packages);

            signer.save_dkg_round2_package(&task.id, &merged);
        }
        Err(e) => {
            return Err(DKGError(e.to_string()));
        }
    };
```

```
    Ok(())
}
```

Listing 2.3: src/apps/signer/dkg.rs

```
pub fn received_round1_packages(task: &mut DKGTask, packets: BTreeMap<Identifier, keys::dkg::
    round1::Package>, signer: &Signer) {

    // store round 1 packets
    let mut local = signer.get_dkg_round1_package(&task.id).map_or(BTreeMap::new(), |v|v);

    // merge packets with local
    local.extend(packets);
    signer.save_dkg_round1_package(&task.id, &local);

    // let k = local.keys().map(|k| to_base64(&k.serialize()[..])).collect::<Vec<_>>();
    debug!("Received round1 packets: {} {:?}", task.id, local.keys());

    // if DB.insert(format!("dkg-{}-round1", task.id), serde_json::to_vec(&local).unwrap()).
        is_err() {
    //    error!("Failed to store DKG Round 1 packets: {} ", task.id);
    // }

    if task.participants.len() == local.len() {

        info!("Received round1 packets from all participants: {}", task.id);
        match generate_round2_packages(&signer, task, local) {
            Ok(_) => {
                task.round = Round::Round2;
                signer.save_dkg_task(&task);
            }
            Err(e) => {
                task.round = Round::Closed;
                signer.save_dkg_task(&task);
                error!("Failed to generate round2 packages: {} - {:?}", task.id, e);
            }
        }
        return;
    }
}
```

Listing 2.4: src/apps/signer/dkg.rs

```
async fn submit_dkg_address(signer: &Signer) {
    for task in signer.list_dkg_tasks().iter_mut() {
        if task.round != Round::Closed {
            continue;
        }

        if task.submitted {
            continue;
        }

```

```
            let task_id = task.id.replace("dkg-", "").parse().unwrap();
            // submit the vault address to sidechain
            let cosm_msg = MsgCompleteDkg {
                id: task_id,
                sender: signer.config().relayer_bitcoin_address(),
                vaults: task.dkg_vaults.clone(),
                consensus_address: signer.validator_address(),
                signature: signer.get_complete_dkg_signature(task_id, &task.dkg_vaults),
            };

            let any = Any::from_msg(&cosm_msg).unwrap();
            match send_cosmos_transaction(signer.config(), any).await {
                Ok(resp) => {
                    let tx_response = resp.into_inner().tx_response.unwrap();
                    if tx_response.code == 0 {
                        task.submitted = true;
                        signer.save_dkg_task(task);

                        info!("Sent dkg vault: {:?}", tx_response.txhash);
                        continue;
                    }

                    error!("Failed to send dkg vault: {:?}", tx_response);
                },
                Err(e) => {
                    error!("Failed to send dkg vault: {:?}", e);
                },
            };
        };

    }
```

Listing 2.5: src/apps/signer/tick.rs

Impact If function `frost::keys::dkg::part2()` fails, an empty vault address will be submitted to the Side Chain.

Suggestion Introduce an additional `Complete` status for tasks, indicating that the Vault address has been successfully generated. Only tasks in `Complete` status should be submitted to the Side Chain.

2.1.3 Lack of task existence check in function `received_sign_message()`

Severity Low

Status Confirmed

Introduced by `Version 1` (Shuttler)

Description In the `received_sign_message()` function, during the processing of the first stage (i.e., Round1) of signing messages, the protocol extracts `task_id` from the message. If the commit for the task does not exist, it invokes `remote_commitments.insert(*index, incoming.clone())` to establish a new index. However, there is no validation to ensure that the `task_id` exists within the system. This oversight poses a risk where attackers, either by compromising trusted participants or acting as malicious users, could inject a large number of invalid task messages into the network. This can overwhelm other nodes and lead to a DoS attack.

```
pub fn received_sign_message(ctx: &mut Context, signer: &Signer, msg: SignMesage) {

    // Ensure the message is not forged.
    match PublicKey::from_slice(&msg.sender.serialize()) {
        Ok(public_key) => {
            let raw = serde_json::to_vec(&msg.package).unwrap();
            let sig = Signature::from_slice(&msg.signature).unwrap();
            if public_key.verify(&raw, &sig).is_err() {
                debug!("Reject, untrusted package from {:?}", msg.sender);
                return;
            }
        }
        Err(_) => return
    }

    // Ensure the message is from the participants
    if !mem_store::is_peer_trusted_peer(&msg.sender, signer) {
        return
    }

    let task_id = msg.task_id.clone();
    let first = 0;

    match msg.package {
        SignPackage::Round1(commitments) => {

            let mut remote_commitments = signer.get_signing_commitments(&task_id);
            // return if msg has received.
            if let Some(exists) = remote_commitments.get(&first) {
                if exists.contains_key(&msg.sender) {
                    return
                }
            }

            // merge received package
            commitments.iter().for_each(|(index, incoming)| {
                match remote_commitments.get_mut(index) {
                    Some(existing) => {
                        existing.extend(incoming);
                    },
                    None => {                    remote_commitments.insert(*index, incoming.
                        clone());
                    },
                }
            });

            signer.save_signing_commitments(&task_id, &remote_commitments);

            try_generate_signature_shares(ctx, signer, &task_id);
```

```
        },
```

Listing 2.6: src/apps/signer/sign.rs

Impact The system may store a large number of invalid tasks and commits, consuming storage resources. In extreme cases, it could occupy legitimate `task_id`, causing the processing of those tasks to fail.

Suggestion Revise the round1 logic to include a check for task existence.

Feedback from the project We are aware of this issue. However, verifying the legitimacy of each task requires on-chain interactions, which increases the associated costs. Therefore, we modified the data structure used in the round1 phase to mitigate the impact caused by malicious nodes.

2.1.4 Insufficient check on block headers

Severity High

Status Fixed in `Version 2` (Side Chain)

Introduced by `Version 1` (Side Chain)

Description The Side chain allows accounts to submit block headers of the BTC chain to the Side chain for validating deposit and withdrawal transactions from users. However, there is no restriction on the accounts that submit block headers. It means that arbitrary accounts can submit block headers to the Side chain, allowing malicious actors to exploit the insufficient validation of the block headers.

Specifically, to be compatible with the reorg feature of the BTC chain, Side chain allows users to submit a series of block headers whose starting height has already been submitted. In this case, the Side chain will check if the reorg is valid to compare the difficulty of the newly submitted block headers with the previously submitted ones.

However, the comparison is not correct. The Side chain only compares the difficulty of the new block header at the starting height with the submitted one at the same height. A malicious user only needs to generate a new valid block within the valid reorg timeframe defined by the Side chain to replace the valid one. Once this malicious action succeeds, the legitimate block header submissions will fail because no valid block following the malicious block will be generated on the BTC chain.

```
func (k Keeper) SetBlockHeaders(ctx sdk.Context, blockHeaders []*types.BlockHeader) error {
    store := ctx.KVStore(k.storeKey)

    // first check if some block header already exists
    for _, header := range blockHeaders {
        if store.Has(types.BtcBlockHeaderHashKey(header.Hash)) {
            // return no error
            return nil
        }
    }

```

```
    params := k.GetParams(ctx)

    // get the best block header
    best := k.GetBestBlockHeader(ctx)

    for _, header := range blockHeaders {
        // validate the block header
        if err := header.Validate(); err != nil {
            return err
        }

        // check if the previous block exists
        if !store.Has(types.BtcBlockHeaderHashKey(header.PreviousBlockHash)) {
            return errorsmod.Wrap(types.ErrInvalidBlockHeader, "previous block does not exist")
        }

        // check the block height
        prevBlock := k.GetBlockHeader(ctx, header.PreviousBlockHash)
        if header.Height != prevBlock.Height+1 {
            return errorsmod.Wrap(types.ErrInvalidBlockHeader, "incorrect block height")
        }

        // check whether it's next block header or not
        if best.Hash != header.PreviousBlockHash {
            // check if the reorg depth exceeds the safe confirmations
            if best.Height-header.Height+1 > uint64(params.Confirmations) {
                return types.ErrInvalidReorgDepth
            }

            // check if the new block header has more work than the old one
            oldNode := k.GetBlockHeaderByHeight(ctx, header.Height)
            worksOld := blockchain.CalcWork(types.BitsToTargetUint32(oldNode.Bits))
            worksNew := blockchain.CalcWork(types.BitsToTargetUint32(header.Bits))
            if sdk.GetConfig().GetBtcChainCfg().Net == wire.MainNet && worksNew.Cmp(worksOld)
                 <= 0 || worksNew.Cmp(worksOld) < 0 {
                return types.ErrForkedBlockHeader
            }

            // remove the block headers after the forked block header
            // and consider the forked block header as the best block header
            for i := header.Height; i <= best.Height; i++ {
                ctx.Logger().Info("Removing block header: ", i)
                thash := k.GetBlockHashByHeight(ctx, i)
                store.Delete(types.BtcBlockHeaderHashKey(thash))
                store.Delete(types.BtcBlockHeaderHeightKey(i))
            }
        }

        // set the block header
        k.SetBlockHeader(ctx, header)

        // update the best block header
        best = header
```

```
    }

    // set the best block header
    k.SetBestBlockHeader(ctx, best)

    return nil
}
```

Listing 2.7: x/btcbridge/keeper/keeper.go

Impact Malicious block headers can be submitted and recorded, rendering the Side protocol broken.

Suggestion Add sufficient access control for the `SubmitBlockHeaders()` function.

2.1.5 Potential nonce reuse during signature generation process of Round 1

Severity High

Status Fixed in `Version 2` (Shuttler)

Introduced by `Version 1` (Shuttler)

Description In each task's signature generation process, the nonce (i.e., `signing_commitments`) is created during the first call to function `generate_nonce_and_commitment_by_address()` and stored locally. This commitment is then broadcasted to other nodes. When a node receives a commitment, it stores it and tries to run function `try_generate_signature_shares()`. If the commitments received meet the required threshold, the node generates and sends out signature shares.

Consider a network with 10 nodes where a minimum of 7 commitments is needed to proceed. If some nodes (malicious) withhold their commitments or heartbeat, leaving exactly 7 active, the honest nodes will generate a `signature_share` upon receiving these 7 commitments. These malicious nodes might later send their withheld commitments, pushing the total received commitments to between 8 and 10. This causes the honest nodes to generate additional rounds of `signature_share`.

The initial nonce, generated only once, is reused for the same task in multiple rounds. This reuse can expose the `private_key_share` of honest nodes to attacks, as malicious nodes can exploit the repeated use of the same nonce for different commitment rounds.

```
fn generate_commitments(ctx: &mut Context, signer: &Signer, task: &SignTask) {

    if task.status == Status::CLOSE {
        return
    }

    let mut nonces = BTreeMap::new();
    let mut commitments = BTreeMap::new();
    //let mut commitments = signer.get_signing_commitments(&task.id);

    task.inputs.iter().for_each(|(index, input)| {
        if let Some((nonce, commitment)) = generate_nonce_and_commitment_by_address(&input.
            address, signer) {
```

```
            nonces.insert(*index, nonce);
            commitments.insert(*index, (signer.identifier().clone(), commitment));
        }
    });

     // Save nonces to local storage.
    signer.save_signing_local_variable(&task.id, &nonces);

    // Publish commitments to other pariticipants
    let mut msg = SignMesage {
        task_id: task.id.clone(),
        package: SignPackage::Round1(commitments),
        nonce: now(),
        sender: signer.identifier().clone(),
        signature: vec![],
    };
    broadcast_signing_packages(ctx, signer, &mut msg);

    received_sign_message(ctx, signer, msg);
}
```

Listing 2.8: src/protocols/sign.rs

```
pub fn try_generate_signature_shares(swarm: &mut Swarm<TSSBehaviour>, signer: &Signer, task_id: &str) {

    // Ensure the task exists locally to prevent forged signature tasks.
    let mut task = match signer.get_signing_task(task_id) {
        Some(t) => t,
        None => return,
    };

    let stored_nonces = signer.get_signing_local_variable(&task.id);
    if stored_nonces.len() == 0 {
        return;
    }
    let stored_remote_commitments = signer.get_signing_commitments(&task.id);

    let mut broadcast_packages = BTreeMap::new();
    for (index, input) in &task.inputs {

        // filter packets from unknown parties
        if let Some(keypair) = signer.get_keypair_from_db(&input.address) {

            let mut signing_commitments = match stored_remote_commitments.get(&index) {
                Some(e) => e.clone(),
                None => return
            };

            sanitize( &mut signing_commitments, &keypair.pub_key.verifying_shares().keys().map(|k| k).collect::<Vec<_>>());

            let received = signing_commitments.len();
```

```
        if received < keypair.priv_key.min_signers().clone() as usize {
            return
        }
        // Only check the first one, because all inputs are in the same package
        if *index == 0 {
            let participants = keypair.pub_key.verifying_shares().keys().collect::<Vec<_>>();
            let alive = mem_store::count_task_participants(&task_id);

            debug!("Commitments {} {}/[{},{}]", &task.id[..6], received, alive.len(), participants.len());

            if !(received == participants.len() || received == alive.len()) {
                return
            }
            task.participants = alive;
            signer.save_signing_task(&task);
        }

        let signing_package = frost::SigningPackage::new(
            signing_commitments,
            frost::SigningTarget::new(
                &input.sig_hash,
                frost::SigningParameters{
                    tapscript_merkle_root: match keypair.tweak {
                        Some(tweak) => Some(tweak.to_byte_array().to_vec()),
                        None => None,
                    },
                }
            ));

        let signer_nonces = match stored_nonces.get(&index) {
            Some(d) => d,
            None => {
                debug!("not found local nonce for input {index}");
                return
            },
        };

        let signature_shares = match frost::round2::sign(
            &signing_package, signer_nonces, &keypair.priv_key
        ) {
            Ok(shares) => shares,
            Err(e) => {
                error!("Error: {:?}", e);
                return;
            }
        };

        let mut my_share = BTreeMap::new();
        my_share.insert(signer.identifier().clone(), signature_shares);

        // broadcast my share
```

```
                broadcast_packages.insert(index.clone(), my_share.clone());

            };
        };

        if broadcast_packages.len() == 0 {
            return;
        }

        let mut msg = SignMesage {
            task_id: task.id.clone(),
            package: SignPackage::Round2(broadcast_packages),
            nonce: now(),
            sender: signer.identifier().clone(),
            signature: vec![],
        };

        publish_signing_package(swarm, signer, &mut msg);

        received_sign_message(swarm, signer, msg);
    }
```

Listing 2.9: src/protocols/sign.rs

Impact The shares of the signing key for participants can be revealed due to nonce reuse attacks.

Suggestion Establish the list of participants for the signing request before starting Round 1.

2.1.6 Lack of duplication checks for participants during DKG initiation

Severity Low

Status Fixed in `Version 2` (Side Chain)

Introduced by `Version 1` (Side Chain)

Description The Side Chain initiates the DKG process (i.e., via the `InitiateDKG()` function) by submitting a governance proposal with selected participants. These proposed participants are required to complete the `DKGRequest` for the vault update. Specifically, the `handleDKGRequests()` function in the `abci.go` file verifies that the number of received `completionRequests` is equal to the number of proposed participants (i.e., `req.Participants`). This check ensures that all participants have completed the `DKGRequest`. However, there is no duplication check for the selected participants during the DKG initiation, which could potentially lead to the generation of an invalid `DKGRequest`. As a result, this invalid `DKGRequest` cannot be completed, requiring the team to restart the DKG initiation.

```
func (m msgServer) InitiateDKG(goCtx context.Context, msg *types.MsgInitiateDKG) (*types.
    MsgInitiateDKGResponse, error) {
    if m.authority != msg.Authority {
        return nil, errorsmod.Wrapf(govtypes.ErrInvalidSigner, "invalid authority; expected %s,
             got %s", m.authority, msg.Authority)
    }
```

```go

    if err := msg.ValidateBasic(); err != nil {
        return nil, err
    }
```

Listing 2.10: side/x/btcbridge/keeper/msg_server.go

```go
func (m *MsgInitiateDKG) ValidateBasic() error {
    if _, err := sdk.AccAddressFromBech32(m.Authority); err != nil {
        return errorsmod.Wrap(err, "invalid authority address")
    }

    if len(m.Participants) == 0 || m.Threshold == 0 || m.Threshold > uint32(len(m.Participants)
        ) {
        return ErrInvalidDKGParams
    }

    for _, p := range m.Participants {
        if len(p.Moniker) > stakingtypes.MaxMonikerLength {
            return ErrInvalidDKGParams
        }

        if _, err := sdk.ValAddressFromBech32(p.OperatorAddress); err != nil {
            return errorsmod.Wrap(err, "invalid operator address")
        }

        if _, err := sdk.ConsAddressFromHex(p.ConsensusAddress); err != nil {
            return errorsmod.Wrap(err, "invalid consensus address")
        }
    }
```

Listing 2.11: side/x/btcbridge/types/msg_server.go

```go
func handleDKGRequests(ctx sdk.Context, k keeper.Keeper) {
    pendingDKGRequests := k.GetPendingDKGRequests(ctx)

    for _, req := range pendingDKGRequests {
        // check if the DKG request expired
        if !ctx.BlockTime().Before(*req.Expiration) {
            req.Status = types.DKGRequestStatus_DKG_REQUEST_STATUS_TIMEDOUT
            k.SetDKGRequest(ctx, req)

            continue
        }

        // handle DKG completion requests
        completionRequests := k.GetDKGCompletionRequests(ctx, req.Id)
        if len(completionRequests) != len(req.Participants) {
            continue
        }
```

Listing 2.12: side/x/btcbridge/module/abci.go

Impact A `DKGRequest` containing duplicate participants can never be completed.

Suggestion Add duplication checks during DKG initiation to prevent the generation of invalid `DKGRequest`s.

2.1.7 DoS due to inconsistent vault versions

Severity High

Status Fixed in `Version 2` (Side Chain)

Introduced by `Version 1` (Side Chain)

Description The function `handleVaultTransfer()` in the file `abci.go` processes the completed `DKGRequest`s at the end of each block to transfer assets (i.e., BTC and Runes) from the old vaults to the newly generated vaults. For each completed `DKGRequest`, the function first extracts the source vault version (i.e., `dkgVaultVersion - 1`) and the destination vault version (i.e., the global latest version returned by the function `GetLatestVaultVersion()`). It then proceeds to transfer assets based on the two versions for different vault types.

However, the asset transfer process is problematic because it assumes that the latest BTC and Runes vaults always share the same version (i.e., the version returned by the function `GetLatestVaultVersion()`). For instance, if only the BTC vault is updated during the DKG process, the latest Runes vault remains the old version, which is different from the version of the newly generated BTC vault. This incorrect assumption could lead to a failure of vault transfer. Moreover, if the Runes vaults are further updated with a newly completed DKGRequest, the version of the Runes vault becomes not incremental resulting in frozen funds stored in the old Runes vault. Example details:

- Initial Vault Versions
  - Latest Vault Version: 3
  - RunesVaultVersions: [1, 2, 3]
  - BTCVaultVersions: [1, 2, 3]
- DKGRequest#1, which updates the BTC vault only
  - Latest Vault Version: 4
  - RunesVaultVersions: [1, 2, 3]
  - BTCVaultVersions: [1, 2, 3, 4]
  - The vault transfer process fails due to nonexistent RuneVault with version#4.
- DKGRequest#2, which updates both Runes and BTC vaults
  - Latest Vault Version: 5
  - RunesVaultVersions: [1, 2, 3, 5]
  - BTCVaultVersions: [1, 2, 3, 4, 5]
  - As a result, the assets in the RunesVault#3 can never be transferred to RunesVault#5 and users can never withdraw their assets from RunesVault#3 as well.

```
func handleVaultTransfer(ctx sdk.Context, k keeper.Keeper) {
    completedDKGRequests := k.GetDKGRequests(ctx, types.
        DKGRequestStatus_DKG_REQUEST_STATUS_COMPLETED)

    for _, req := range completedDKGRequests {
        if req.EnableTransfer {
```

```
            completions := k.GetDKGCompletionRequests(ctx, req.Id)
            dkgVaultVersion, _ := k.GetVaultVersionByAddress(ctx, completions[0].Vaults[0])

            sourceVersion := dkgVaultVersion - 1
            destVersion := k.GetLatestVaultVersion(ctx)
```

Listing 2.13: side/x/btcbridge/keeper/abci.go

```
func (k Keeper) UpdateVaults(ctx sdk.Context, newVaults []string, vaultTypes []types.AssetType
    ) {
    params := k.GetParams(ctx)

    version := k.IncreaseVaultVersion(ctx)

    for i, v := range newVaults {
        newVault := &types.Vault{
            Address:  v,
            AssetType: vaultTypes[i],
            Version:  version,
        }

        params.Vaults = append(params.Vaults, newVault)
    }

    k.SetParams(ctx, params)
 }

// IncreaseVaultVersion increases the vault version by 1
func (k Keeper) IncreaseVaultVersion(ctx sdk.Context) uint64 {
    store := ctx.KVStore(k.storeKey)

    version := k.GetLatestVaultVersion(ctx)

    store.Set(types.VaultVersionKey, sdk.Uint64ToBigEndian(version+1))

    return version + 1
}

// GetLatestVaultVersion gets the latest vault version
func (k Keeper) GetLatestVaultVersion(ctx sdk.Context) uint64 {
    store := ctx.KVStore(k.storeKey)

    bz := store.Get(types.VaultVersionKey)
    if bz != nil {
        return sdk.BigEndianToUint64(bz)
    }

    return 0
}
```

Listing 2.14: side/x/btcbridge/keeper/tss.go

```
func (k Keeper) TransferVault(ctx sdk.Context, sourceVersion uint64, destVersion uint64,
    assetType types.AssetType, psbts []string, targetUtxoNum uint32) error {
```

```
    sourceVault := k.GetVaultByAssetTypeAndVersion(ctx, assetType, sourceVersion)
    if sourceVault == nil {
        return types.ErrVaultDoesNotExist
    }

    destVault := k.GetVaultByAssetTypeAndVersion(ctx, assetType, destVersion)
    if destVault == nil {
        return types.ErrVaultDoesNotExist
    }
```

Listing 2.15: side/x/btcbridge/keeper/tss.go

Impact The vault transfer for a completed DKGRequest updating only one vault would fail.

Suggestion Assign the correct destination version for different vault types during the vault transfer process.

2.1.8 Lack of error handling logic in function `scan_vault_txs()`

Severity Low

Status Fixed in `Version 2` (Shuttler)

Introduced by `Version 1` (Shuttler)

Description The `scan_vault_txs()` function calls the `scan_vault_txs_by_height()` function to scan transactions at a specific block height on the BTC chain. However, if the execution of `scan_vault_txs_by_height()` fails, the `scan_vault_txs()` function does not contain any error-handling logic. As a result, it proceeds to save the current block height regardless of the failure. Consequently, during the next scan, the process will attempt to scan the block at `height + 1` and skip the block where the error occurred.

```
pub async fn scan_vault_txs(relayer: &Relayer) {
    let interval = relayer.config().loop_interval;
    let height = get_last_scanned_height(relayer ) + 1;

    let side_tip =
        match client_side::get_bitcoin_tip_on_side(&relayer.config().side_chain.grpc).await {
            Ok(res) => res.get_ref().height,
            Err(e) => {
                error!("Failed to get tip from side chain: {}", e);
                return;
            }
        };

    let confirmations = client_side::get_confirmations_on_side(&relayer.config().side_chain.
        grpc).await;
    if height > side_tip - confirmations + 1 {
        debug!("No new txs to sync, height: {}, side tip: {}, sleep for {} seconds...", height,
             side_tip, interval);
        return;
    }

    debug!("Scanning height: {:?}, side tip: {:?}", height, side_tip);
```

```rust
        scan_vault_txs_by_height(relayer, height).await;
        save_last_scanned_height(relayer, height);
    }
```

Listing 2.16: src/apps/relayer/tick.rs

Impact The lack of error handling in `scan_vault_txs()` may result in a relayer node failing to submit transactions related to the vault accounts.

Suggestion Add error handling logic for the failure of function `scan_vault_txs_by_height()` execution.

2.2 Recommendations

2.2.1 Lack of error handling logic in function `submit_signature()`

Status Confirmed

Introduced by `Version 1` (Side Chain)

Description In the `get_signing_request_by_txid()` function, an error message is returned when the query client connection fails. However, in the `submit_signatures()` function, the handling of its return value only considers success and empty cases, without addressing error returns. This could lead to a failure to promptly capture errors when the connection fails.

```rust
pub async fn get_signing_request_by_txid(host: &str, txid: String) -> Result<Response<
    QuerySigningRequestByTxHashResponse>, Status> {
    let mut btc_client = match BtcQueryClient::connect(host.to_string()).await {
        Ok(client) => client,
        Err(e) => {
            return Err(Status::cancelled(format!("Failed to create btcbridge query client: {}",
                e)));
        }
    };

    btc_client.query_signing_request_by_tx_hash(QuerySigningRequestByTxHashRequest {
        txid,
    }).await
}
```

Listing 2.17: src/helper/client_side.rs

```rust
pub async fn submit_signatures(psbt: Psbt, signer: &Signer) {

    // broadcast to bitcoin network
    let signed_tx = psbt.clone().extract_tx().expect("failed to extract signed tx");

    let host = signer.config().side_chain.grpc.clone();
    let txid = signed_tx.compute_txid().to_string();
    if let Ok(response) = get_signing_request_by_txid(&host, txid.clone()).await {
        match response.into_inner().request {
            Some(request) => if request.status != SigningStatus::Pending as i32 {
            debug!("Other participant has broadcasted. {txid}",);
```

```
            return;
            },
            None => return,
        };
    };

    match signer.bitcoin_client.send_raw_transaction(&signed_tx) {
        Ok(txid) => {
            info!("PSBT broadcasted to Bitcoin: {}", txid);
        }
        Err(err) => {
            error! ("Failed to broadcast PSBT: {:?}, err: {:?}", signed_tx.compute_txid(), err)
                ;
            // return;
        }
    }

    let psbt_bytes = psbt.serialize();
    let psbt_base64 = to_base64(&psbt_bytes);

    // submit signed psbt to side chain
    let msg = MsgSubmitSignatures {
        sender: signer.config().relayer_bitcoin_address(),
        txid: signed_tx.compute_txid().to_string(),
        psbt: psbt_base64,
    };

    let any = Any::from_msg(&msg).unwrap();
    match send_cosmos_transaction(signer.config(), any).await {
        Ok(resp) => {
            let tx_response = resp.into_inner().tx_response.unwrap();
            if tx_response.code != 0 {
                error!("Failed to submit signatures: {:?}", tx_response);
                return
            }
            info!("Submitted signatures: {:?}", tx_response.txhash);
        },
        Err(e) => {
            error!("Failed to submit signatures: {:?}", e);
        },
    };
    // send message to the network
}
```

Listing 2.18: src/apps/signer/sign.rs

Suggestion Add error handling logic for returned error messages.

2.2.2 Fix potential panics

Status Fixed in `Version 2` (FROST)

Introduced by `Version 1` (FROST)

Description In the function `hasher_to_scalar`, a potential panic may occur if the calculated hash is larger than the group's order.

```
/// Digest the hasher to a Scalar
fn hasher_to_scalar(hasher: Sha256) -> Scalar {
    let sp = ScalarPrimitive::new(U256::from_be_slice(&hasher.finalize())).unwrap();
    Scalar::from(&sp)
}
```

Listing 2.19: frost-secp256k1-tr/src/lib.rs

Suggestion Use `Scalar::reduce` to manage edge cases of hash calculations.

2.2.3 Remove unused code

Status Fixed in `Version 2` (Side Chain)

Introduced by `Version 1` (Side Chain)

Description There are several functions that are not used in the Side Chain. It is recommended to remove unused or deprecated code to maintain code clarity. Below is the list of unused functions:

- side/x/keeper/params.go
  - `EnableBridge()`
  - `DisableBridge()`
- side/x/keeper/withdraw.go
  - `NewSigningRequest()`

Suggestion Remove unused functions for clarity.

2.3 Notes

2.3.1 Potential centralization risk

Introduced by `Version 1` (Side Chain)

Description In the Side Chain, multiple privileged functions (e.g., `UpdateTrustedOracle()`, `UpdateTrustedNonBtcRelayers()`, and `SubmitFeeRate()`) are used to set or update critical configurations, which can lead to potential centralization risks. For example, the function `updateTrustedOracle()` allows a "trusted" oracle to reset the list of trusted oracles.

Additionally, some functions (e.g., `InitiateDKG()` and `UpdateParams()`) may be vulnerable to governance attacks. For instance, the Side Chain allows anyone to submit a governance proposal to initiate the DKG process via the function `InitiateDKG()` function. If malicious users gain majority governance power, they could initiate the DKG process with arbitrarily selected participants, disrupting the process and potentially leading to unexpected fund losses.

2.3.2 Trusted participants in the DKG process

Introduced by `Version 1` (FROST & Shuttler)

Description During the audit process, we assume that all participants in the DKG process are trusted and refrain from any malicious activities. If this assumption is violated, the entire protocol becomes vulnerable to various attacks, including but not limited to: Malicious commitment broadcast. In Round 1 of the DKG process, each participant broadcasts polynomial commitments to the others. If a malicious participant sends different commitments to different participants, the DKG progress will succeed while the later signing tasks will not generate valid signature shares, which leads to fund losses.

2.3.3 Frozen protocol fees

Introduced by `Version 1` (Side Chain)

Description In the Side Chain, the protocol fee collector is set to the governance module account by default. However, the protocol does not implement any functions to handle the collected fee. Without implementing fee collection functions or updating the protocol fee collector to an account accessible by the team, the collected fees could potentially become inaccessible, resulting in frozen funds.

2.3.4 Rune verification reliance on external mechanisms

Introduced by `Version 1` (Shuttler)

Description In Shuttler, this component relies on external mechanisms to manage and verify rune-related transactions. This includes the logic for retrieving rune data by ID and for retrieving and validating outputs. If the external mechanisms fail or the external data is tampered with, it may result in invalid validation outcomes.

2.3.5 Potential DoS due to expired fee rate

Introduced by `Version 1` (Side Chain)

Description In the Side Chain, most functionalities (e.g., withdraw, transferVault) could face a DoS issue if the fee rate is inactive. By default, the `FeeRateValidityPeriod` is only 100 blocks. Therefore, the team must ensure that the fee rate is timely updated to prevent potential DoS issues by an expired fee rate.

2.3.6 Potential DoS due to excessive unrelated commitments in Round 1 message

Introduced by `Version 1` (Shuttler)

Description While issue-3 (Lack of task existence check in function `received_sign_message()`) could be addressed by verifying the legitimacy of each received task on-chain, such an approach would significantly increase system overhead. To mitigate the risk, the project team improved the data structure. The original design, which allowed each participant to include multiple commitments in a single broadcasted data packet (`BTreeMap<Index, BTreeMap<Identifier, SigningCommitments>>`), was revised so that each packet corresponds to only one commitment

(`BTreeMap<Index, (Identifier, SigningCommitments)>`). However, this modification only reduces the impact of a single attack. Malicious nodes can still achieve DoS attacks by constructing and sending a large number of commitments related to unrelated tasks.

2.3.7 Private keys and passwords are stored in plaintext format

Introduced by `Version 1` (Shuttler)

Description During the initialization of Shuttler, the `default()` function in the `config/mod.rs` module is invoked to initialize the RPC configuration file. However, the implementation stores sensitive information in plaintext format without any encryption or protective measures. If an attacker gains access to the system, they could easily access this private data.

1. Validator private keys are generated and directly written to the file system in plaintext JSON format using `fs::write()`.
2. The RPC default configuration contains hardcoded usernames and passwords within the configuration program.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS