Stacks sBTC Withdrawal

Security Assessment

April 10th, 2025 — Prepared by OtterSec

Samuel Bétrisey sam@osec.io

Renato Eugenio Maria Marziano renato@osec.io

Robert Chen r@osec.io

Table of Contents

Executive Summary **2**

Overview 2

Key Findings 2

Scope **3**

Findings **4**

Vulnerabilities **5**

OS-SBT-ADV-00 | Exposed Event Port 6

OS-SBT-ADV-01 | DOS Due to Unchecked Stacks Transaction Fee 7

OS-SBT-ADV-02 | Missing Nonce Validation 8

OS-SBT-ADV-03 | Inconsistent Dust Limit Enforcement 9

OS-SBT-ADV-04 | Threshold Signature Abuse via Signature Withholding 11

General Findings **12**

OS-SBT-SUG-00 | Unauthorized Withdrawal Due to Utilization of Tx-Sender 13

OS-SBT-SUG-01 | Disconnecting P2P Connections with Removed Peers 14

OS-SBT-SUG-02 | Code Refactoring 15

Appendices

Vulnerability Rating Scale **17**

Procedure **18**

01 — Executive Summary

Overview

Stacks Network engaged OtterSec to assess the `sBTC` implementation, focusing specifically on the withdrawals functionality. This assessment was conducted between February 18th and April 2nd, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 8 findings throughout this audit engagement.

In particular, we identified a vulnerability where exposing the Stacks event-observer port enables attackers to submit fake Stacks events to the signer, resulting in unauthorized modifications of the signer's database without verification (OS-SBT-ADV-00). Additionally, the lack of nonce validation in the transaction signing process allows a malicious coordinator to request signatures for transactions with inflated nonces, potentially causing future transactions to fail when broadcast (OS-SBT-ADV-02). Furthermore, the dust limit validation is inconsistent across deposits and withdrawals, allowing deposits at the dust limit but rejecting withdrawals of the same amount, resulting in locking of user funds (OS-SBT-ADV-03).

We also made recommendations for modifying the codebase for improved functionality, efficiency, and robustness (OS-SBT-SUG-02), and suggested ensuring the system disconnects removed peers and handles all other message types appropriately (OS-SBT-SUG-01). We further advised utilizing contract caller instead of tx-sender while initiating a withdrawal request to prevent malicious contracts from exploiting post-conditions to withdraw a user's sBTC (OS-SBT-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/stacks-network/sbtc. This audit was performed against 475f845.

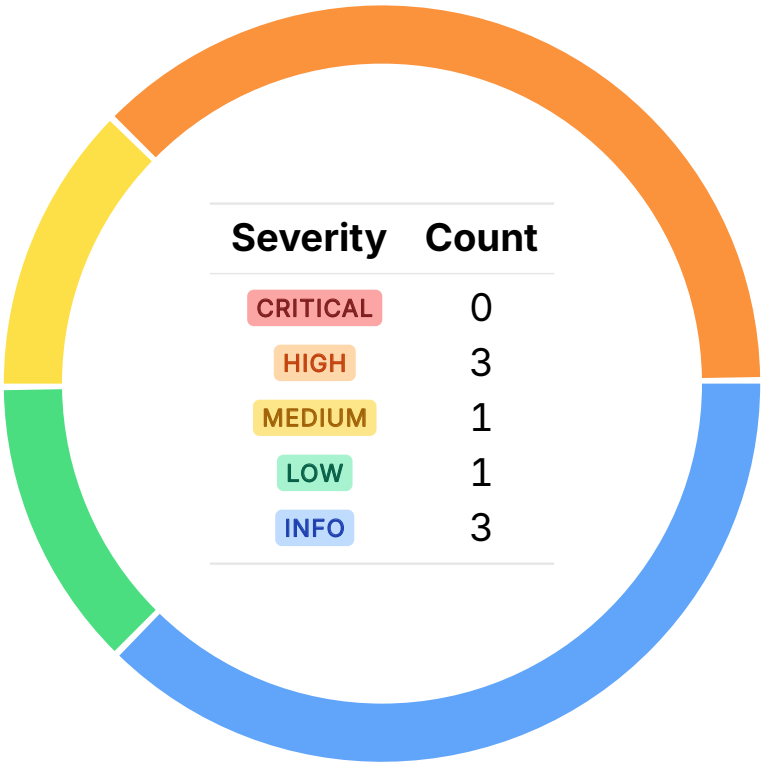
A brief description of the program is as follows:

Name	Description
sBTC	The Stacks sBTC module enables Bitcoin to be represented as a SIP-010 token (sBTC) on the Stacks blockchain, allowing BTC holders to access smart contracts and DeFi.

03 — Findings

Overall, we reported 8 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	3
MEDIUM	1
LOW	1
INFO	3

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-SBT-ADV-00	HIGH	RESOLVED	Exposing the Stacks event-observer port allows potential attackers to submit fake Stacks events to the signer, resulting in unauthorized modifications of the signer's database without verification.
OS-SBT-ADV-01	HIGH	RESOLVED	Transaction signers do not validate the `tx_fee`, allowing a malicious coordinator to set excessively high fees that may drain the multisig wallet, preventing future transactions until it is refilled.
OS-SBT-ADV-02	MEDIUM	RESOLVED	The lack of nonce validation in the transaction signing process allows a malicious coordinator to request signatures for transactions with inflated nonces, potentially causing future transactions to fail when broadcast.
OS-SBT-ADV-03	LOW	RESOLVED	The dust limit validation is inconsistent across deposits and withdrawals, allowing deposits at the dust limit but rejecting withdrawals of the same amount, resulting in the locking of user funds.
OS-SBT-ADV-04	HIGH	RESOLVED	A malicious signer may withhold their final signature share, collect others', and later finalize and broadcast a `BTC` withdrawal after the request is rejected and `sBTC` is unlocked.

Exposed Event Port HIGH OS-SBT-ADV-00

Description

The `docker-compose.yml` configuration exposes port `8801` on the Stacks sBTC signer container to all interfaces in the default mainnet Docker setup. Moreover, the Stacks event-observer does not validate the source of incoming events. This poses a risk, as Docker iptables rules generally take precedence, and signers are aware of each other's IP addresses, allowing potential P2P communication. Thus, without an added firewall, an attacker may submit `new_block` notifications with fake Stacks events, which will be accepted and added to the signer's database without verification. If enough signers are vulnerable to this attack, it is possible to create fake withdrawals and steal the BTC controlled by the signers.

Remediation

Restrict the port to trusted sources, and implement source validation on the event-observer to ensure that incoming events originate from trusted sources.

Patch

Resolved in 8d2e6de.

DOS Due to Unchecked Stacks Transaction Fee HIGH OS-SBT-ADV-01

Description

In `transaction_signer::handle_stacks_transaction_sign_request`, the multisig transaction is constructed without validating the `tx_fee` parameter. The signers do not validate the `tx_fee` of Stacks transactions, and `request.tx_fee` is passed directly into the transaction creation process. This enables a malicious transaction coordinator to set an excessively high transaction fee (`tx_fee`), which the other signers will sign, draining the multisig wallet's `STX` balance. This will leave the multisig wallet with zero `STX`, preventing any future transactions until the multisig wallet is funded with `STX`.

>_ *sbtc/signer/src/transaction_signer.rs* RUST

```rust
async fn handle_stacks_transaction_sign_request(
    &mut self,
    request: &StacksTransactionSignRequest,
    chain_tip: &model::BitcoinBlockRef,
    origin_public_key: &PublicKey,
) -> Result<(), Error> {
   [...]
   let multi_sig = MultisigTx::new_tx(&request.contract_tx, &wallet, request.tx_fee);
   let txid = multi_sig.tx().txid();
   [...]
}
```

Remediation

Introduce a global fee limit per transaction, such that the signers reject any transaction where `tx_fee` exceeds this limit.

Patch

Resolved in 7da327c.

Missing Nonce Validation MEDIUM OS-SBT-ADV-02

Description

Currently, the nonce of Stacks transactions is not validated by signers in `handle_stacks_transaction_sign_request` in `transaction_signer`. This implies that the nonce provided in `StacksTransactionSignRequest` is directly utilized to set the nonce for signing the transaction. Thus, a malicious transaction coordinator may craft a request to sign a transaction with a higher nonce and broadcast it later, rendering a future stacks transaction to fail.

>_ *sbtc/signer/src/transaction_signer.rs* RUST

```rust
async fn handle_stacks_transaction_sign_request(
    &mut self,
    request: &StacksTransactionSignRequest,
    chain_tip: &model::BitcoinBlockRef,
    origin_public_key: &PublicKey,
) -> Result<(), Error> {
   [...]
   // We need to set the nonce in order to get the exact transaction
   // that we need to sign.
   let wallet = SignerWallet::load(&self.context, &chain_tip.block_hash).await?;
   wallet.set_nonce(request.nonce);
   [...]
}
```

Remediation

Ensure the signers validate that the provided nonce is greater by a single value before setting the nonce and signing the transaction.

Patch

Fix in progress.

Inconsistent Dust Limit Enforcement LOW OS-SBT-ADV-03

Description

The vulnerability arises due to inconsistent validation of the dust limit across different parts of the system. The dust limit is a threshold below which a transaction amount is considered too small to be processed meaningfully. Currently, the system allows a deposit if the amount is greater than or equal to the `dust-limit`, but in `sbtc-withdrawa::initiate-withdrawal-request`, a withdrawal is only allowed if the amount is strictly greater than the dust limit. The Rust-based signer logic in `WithdrawalRequestReport::validate` allows withdrawals when the amount is equal to or greater than the dust limit.

>_ contracts/contracts/sbtc-deposit.clar RUST

```
;; Accept a new deposit request
(define-public (complete-deposit-wrapper ([...])
    (let
            (
                    (current-signer-data (contract-call? .sbtc-registry
                        ↪ get-current-signer-data))
                    (replay-fetch (contract-call? .sbtc-registry get-deposit-status txid
                        ↪ vout-index))
            )
        [...]
        ;; Check that amount is greater than dust limit
        (asserts! (>= amount dust-limit) ERR_LOWER_THAN_DUST)
        [...]
    )
)
```

Thus, in the existing implementation, a user may successfully deposit an amount exactly equal to the dust limit. However, when they attempt to withdraw the same amount, the contract rejects it. This effectively locks the funds in the system, as withdrawals below or equal to the dust limit are not permitted.

>_ contracts/contracts/sbtc-withdrawal.clar RUST

```
(define-public (initiate-withdrawal-request (amount uint)
        [...]
        (begin
                (try! (contract-call? .sbtc-token protocol-lock (+ amount max-fee) tx-sender
                    ↪ withdraw-role))
                (asserts! (> amount DUST_LIMIT) ERR_DUST_LIMIT)
            [...]
        )
)
```

Remediation

Ensure uniform conditions for deposits and withdrawals by aligning all dust limit checks across the system.

Patch

Fix in progress.

Threshold Signature Abuse via Signature Withholding HIGH OS-SBT-ADV-04

Description

A malicious signer may collect signature shares from `T-1` honest parties but intentionally withhold their own, preventing the completion of a `BTC` withdrawal transaction. Once the on-chain withdrawal request is rejected and the `sBTC` is unlocked, the attacker may then finalize and broadcast the `BTC` transaction utilizing the previously collected shares. However, for this to be possible, there must be only `T-1` parties signing the transaction. Thus, the attack is only feasible if the malicious signer controls at least `N-T+1` participants.

Remediation

Introduce an active withdrawal check to prevent premature rejection of withdrawal requests, ensuring such active withdrawals are not rejected and the coordinator only processes safe-to-reject requests. This prevents malicious signers from broadcasting transactions after `sBTC` is unlocked.

Patch

This issue was already identified by the team and fixed in PR#1411.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-SBT-SUG-00	`initiate-withdrawal-request` utilizes `tx-sender` rather than `contract-caller`, rendering it vulnerable to malicious contracts that exploit post-conditions to withdraw a user's `sBTC`.
OS-SBT-SUG-01	There is a discrepancy between the expected behavior and the current implementation, where removed peers may still send messages that are filtered by the `is_allowed_peer` check.
OS-SBT-SUG-02	Recommendation for modifying the codebase for improved functionality, efficiency, and robustness.

Unauthorized Withdrawal Due to Utilization of Tx-Sender OS-SBT-SUG-00

Description

`initiate-withdrawal-request` currently utilizes `tx-sender` instead of `contract-caller`. This allows a malicious contract to trigger the function on behalf of a user, potentially draining their `sBTC` holdings if they have approved `sBTC` spending in post-conditions. Furthermore, there are additional issues concerning the utilization of `tx-sender`.

>_ *contracts/contracts/sbtc-withdrawal.clar* RUST

```
(define-public (initiate-withdrawal-request (amount uint)
        [...]
        (begin
                (try! (contract-call? .sbtc-token protocol-lock (+ amount max-fee) tx-sender
                    ↪  withdraw-role))
            [...]
        )
)
```

Allowing a contract to transfer a user's sBTC may have a legitimate purpose, but letting a contract initiate a withdrawal does not seem necessary. Also, if the transaction is simulated, the user will see that they will receive locked `sBTC` in exchange. In a social engineering attack, this insight may be utilized to pretend that the user is simply stacking tokens.

Remediation

Utilize `contract-caller` instead of `tx-sender`.

Disconnecting P2P Connections with Removed Peers OS-SBT-SUG-01

Description

When the signing set is updated, the system does not immediately disconnect P2P connections with peers who are removed from the set. This results in a situation where peers who are no longer part of the allowed signer set may still send messages, but these messages are ignored because of the check in `handle_gossipsub_event`. However, the comment indicates that this check may be unnecessary and should be removed.

>_ sbtc/signer/src/network/libp2p/event_loop.rs RUST

```rust
fn handle_gossipsub_event(
    swarm: &mut Swarm<SignerBehavior>,
    ctx: &impl Context,
    event: gossipsub::Event,
) {
    use gossipsub::Event;
    match event {
        Event::Message {
            propagation_source: peer_id,
            message,
            ..
        } => {
            let current_signer_set = ctx.state().current_signer_set();
            // The following check should be unnecessary. In order to
            // receive a message the peer needs to establish a connection,
            // and in order to do that the peer needs to be in the current
            // signer set. When we implement the signing set changing code,
            // we should re-evaluate whether we should remove this check.
            if !current_signer_set.is_allowed_peer(&peer_id) {
                tracing::warn!(%peer_id, "ignoring message from unknown peer");
                return;
            }
            [...]
        }
        [...]
    }
}
```

Additionally, while the `gossipsub` messages are filtered out due to the signer set check, other types of messages, such as `Kademlia` events, may still be processed.

Remediation

Ensure the system disconnects removed peers and handles all other message types appropriately.

Code Refactoring OS-SBT-SUG-02

Description

1. In the current implementation, the transaction coordinator and transaction signer are initialized with a static threshold value that is loaded from the configuration (`config.signer.bootstrap_signatures_required`), which remains unchanged throughout the runtime, potentially creating issues if the threshold needs dynamic adjustments based on evolving conditions.
2. When the signer program starts, it loads the `bootstrap_signing_set` from the configuration. However, the `bootstrap_signing_set` may be outdated because it is loaded at the time of initialization from the configuration and utilizes this until the next Bitcoin block. If the set of valid signers changes, the program will still utilize the outdated list. Thus, these signers may still establish peer-to-peer (P2P) connections with the signer program. It would be appropriate to dynamically check the validity of the signers in `bootstrap_signing_set`.

   >_ sbtc/signer/src/main.rs RUST

   ```rust
   #[tracing::instrument(name = "signer")]
   async fn main() -> Result<(), Box<dyn std::error::Error>> {
       [...]
       // TODO: We should first check "another source of truth" for the current
       // signing set, and only assume we are bootstrapping if that source is
       // empty.
       let settings = context.config();
       for signer in settings.signer.bootstrap_signing_set() {
           context.state().current_signer_set().add_signer(signer);
       }
       [...]
   }
   ```

3. In `complete-individual-withdrawal-helper`, most optional values (`bitcoin-txid`, `output-index`, and `fee`) are checked via `is-some` before unwrapping with `unwrap-panic`. However, `sweep-txid` is directly unwrapped (`unwrap-panic (get sweep-txid withdrawal)`) without a prior validation check. If `sweep-txid` is `none`, calling `unwrap-panic` will result in a panic instead of returning an error with `ERR_WITHDRAWAL_INDEX_PREFIX + index`. Ensure that if `sweep-txid` is missing, the function returns an appropriate error instead of terminating unexpectedly.
4. The `BTC` fee rate is determined by the transaction coordinator but is not independently validated by the signers. While signers do verify that the fee remains within the user-specified maximum for deposits or withdrawals, it is still advisable to explicitly validate the coordinator-proposed Bitcoin/Stacks fees to ensure correctness and prevent potential misuse.

Remediation

Incorporate the above-mentioned refactors.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.