CoinFabrik

Security Audit Report

Stacks - Signer

April 2025

V202411

Executive Summary

CoinFabrik was asked to audit the **Stacks Signer** component for the **Stacks** project.

During this audit we found one high severity issue, one medium severity issue and five low severity issues. Also, one enhancement was proposed.

All the issues were resolved or acknowledged.

Scope

The audited files are from the git repository located at https://github.com/stacks-network/stacks-core.git. The audit is based on the commit `c1a1f50fddcbc11054fae537103423e21221665a`.

The scope for this audit includes and is limited to the following files:

- **`src/chainstate.rs`**: Stacks sortition state and definitions
- **`src/cli.rs:`** CLI subcommand implementation
- **`src/config.rs:`** Config file manager
- **`src/lib.rs:`** CLI definitions
- **`src/main.rs:`** CLI top-level code
- **`src/monitor_signers.rs:`** Observe and report signer behavior
- **`src/runloop.rs:`** Top-level runloop, containing main signer logic
- **`src/signerdb.rs:`** Maintain state of the signer
- **`src/client:`** StackerDB client
- **`src/monitoring:`** Monitoring endpoint implementation
- **`src/v0: Signer implementation`**

No other files in this repository were audited. Its dependencies are assumed to work according to their documentation. Also, no tests were reviewed for this audit.

Findings

In the following table we summarize the security issues we found in this audit. The severity classification criteria and the status meaning are explained below. This table does not include

the enhancements we suggest to implement, which are described in a specific section after the security issues.

Each severity label is detailed in the Severity Classification section. Additionally, the statuses are explained in the Issues Status section.

Id	Title	Severity	Status
HI-01	Replay Attack on Vote	High	**Acknowledged**
ME-01	Incorrect Block Rejection State Management	Medium	**Acknowledged**
LO-01	Memory Leak Of Stale Signers	Low	**Resolved**
LO-02	Insecure Default Contract Creation	Low	**Resolved**
LO-03	Incorrect Reward Cycle In Calculation	Low	**Acknowledged**
LO-04	Insecure transmission of authentication credentials	Low	**Acknowledged**
LO-05	Stale configuration Handling Of Reward Cycle	Low	**Acknowledged**

Critical Severity Issues

No issues found.

High Severity Issues

HI-01 Replay Attack on Vote

Location

- `./src/cli.rs: 182`

Classification

- CWE-294: Authentication Bypass by Capture-replay[1]

[1] https://cwe.mitre.org/data/definitions/294.html

Description

The `VoteInfo::digest` method constructs a message hash without including a nonce/timestamp. This allows signatures to be replayed across voting periods. Attackers can capture a valid signature and resubmit it indefinitely.

We can see in this code:

```
fn digest(&self) -> Sha256Sum {
    let vote_message = TupleData::from_data(vec![
        ("sip".into(), Value::UInt(self.sip.into())),
        ("vote".into(), Value::UInt(self.vote.to_u8().into())),
    ])
    .unwrap();
    let data_domain =
        make_structured_data_domain("signer-sip-voting", "1.0.0", CHAIN_ID_MAINNET);
    structured_data_message_hash(vote_message.into(), data_domain)
```

The absence of a nonce (a unique random value used once) or a timestamp (indicating when the signature was created) means that the same signature can be reused multiple times.

Additionally, because SIP proposals are small consecutive numbers, a malicious operator can pre-generate most possible votes in advance, by signing “yes” and “no” votes, and then storing them for later use.

Recommendation

Incorporate a nonce and a timestamp in the signature generation process, to ensure that each signature is unique and cannot be reused. Also, a mechanism to expire those signatures must be implemented in the verification process too.

Status

Acknowledged. Voters can only cast their vote once, and because the count is manual, any attack can be detected. But in future versions, the recommendation to incorporate a nonce was accepted.

Medium Severity Issues

ME-01 Incorrect Block Rejection State Management

Location

- `./src/v0/signers.rs: 938`

Classification

- CWE-770: Allocation of Resources Without Limits or Throttling[2]

Description

The `check_submitted_block_proposal()` function broadcasts block rejections even when marking the block as locally rejected fails, risking inconsistent node states.

We can see that problem in this code:

```
        let rejection =
            self.create_block_rejection(RejectCode::ConnectivityIssues, &block_info.block);
        if let Err(e) = block_info.mark_locally_rejected() {
            if !block_info.has_reached_consensus() {
                warn!("{self}: Failed to mark block as locally rejected: {e:?}");
            }
        };
        debug!("{self}: Broadcasting a block response to stacks node: {rejection:?}");
        let res = self.stackerdb
            .send_message_with_retry::<SignerMessage>(rejection.into());
```

When a block validation response is not received within the configured timeout (`block_proposal_validation_timeout`), the function attempts to mark the block as locally rejected via `block_info.mark_locally_rejected()`. If this operation fails (e.g., due to database errors or race conditions), the function logs a warning but proceeds to broadcast a rejection message to the network (`self.stackerdb.send_message_with_retry`).

Recommendation

If `mark_locally_rejected()` fails, abort the rejection broadcast to maintain consistency.

Status

Acknowledged. According to the developers, this is the intended behavior.

[2] https://cwe.mitre.org/data/definitions/770.html

Low Severity Issues

LO-01 Memory Leak Of Stale Signers

Location

- `./src/runloop.rs: 451`

Classification

- CWE-401: Missing Release of Memory after Effective Lifetime[3]

Description

The `cleanup_stale_signers` function fails to release resources associated with non-registered signers, potentially leading to memory leaks and impacting application performance over time.

We can see that problem in this code:

```
        for (idx, signer) in &mut self.stacks_signers {
                ........
            if let ConfiguredSigner::RegisteredSigner(signer) = signer {
                if !signer.has_unprocessed_blocks() {
                    debug!("{signer}: Signer's tenure has completed.");
                    to_delete.push(*idx);
                }
            }
        }
```

The function iterates through the collection of signers. For each stale signer, it checks if the signer is a `RegisteredSigner` and whether it has unprocessed blocks. If a signer is stale and has no unprocessed blocks, its index is added to a `to_delete` vector for removal from the `stacks_signers` collection.

However, the function does not account for non-registered signers. If a signer is not a `RegisteredSigner`, it is ignored entirely, and no cleanup or resource release occurs for that signer.

Recommendation

Modify the `cleanup_stale_signers` function to include logic for releasing resources associated with non-registered signers.

[3] https://cwe.mitre.org/data/definitions/401.html

Status

Resolved. Unregistered signers are now correctly released.

LO-02 Insecure Default Contract Creation

Location

- `./src/client/stacks_client.rs: 793`

Classification

- CWE-1188: Initialization of a Resource with an Insecure Default[4]

Description

The `build_unsigned_contract_call_transaction()` function sets `post_condition_mode` to `Allow` by default, disabling critical security checks for state changes.

We can see that problem in this code:

```
    unsigned_tx.anchor_mode = TransactionAnchorMode::Any;
    unsigned_tx.post_condition_mode = TransactionPostConditionMode::Allow;
     unsigned_tx.chain_id = chain_id;
     Ok(unsigned_tx)
```

We can see in the code that the `post_condition_mode` is set to `TransactionPostConditionMode::Allow` by default. This default behavior disables essential security checks that validate state changes after contract execution, leaving transactions vulnerable to exploitation.

Recommendation

Set the default `post_condition_mode` to `TransactionPostConditionMode::Deny` or a more restrictive mode to enforce security checks unless explicitly overridden..

Status

Resolved. The insecure method was removed as it was currently unused.

[4] https://cwe.mitre.org/data/definitions/770.html

LO-03 Incorrect Reward Cycle In Calculation

Location

- `./src/v0/signer.rs: 142`

Classification

- CWE-682: Incorrect calculation[5]

Description

The `process_event` function incorrectly uses `self.reward_cycle` instead of `current_reward_cycle`, potentially leading to the skipping of valid events due to outdated signer's cycle information.

We can see that problem in this code:

```
        let other_signer_parity = (self.reward_cycle + 1) % 2;
        if event_parity == Some(other_signer_parity) {
            Return;
        }
```

In this code, `self.reward_cycle` is used to determine the `other_signer_parity`, which is then compared against the `event_parity`. If the signer's cycle (`self.reward_cycle`) is outdated, it may not accurately reflect the current state of the network, leading to valid events being incorrectly filtered out. This issue could lead to a denial of service (DoS) for the signer, as it may fail to respond to valid events.

Recommendation

To mitigate this vulnerability, it is recommended to modify the process_event function to use `current_reward_cycle` instead of `self.reward_cycle` when computing `other_signer_parity`.

Status

Acknowledged. The parity calculation is always (self.reward_cycle+1)%2 regardless of the current_reward_cycle. This is the correct behavior and thus the issue does not exist.

[5] https://cwe.mitre.org/data/definitions/682.html

LO-04 Insecure transmission of authentication credentials

Location

- `./src/client/stacks_client.rs: 142`

Classification

- CWE-319: Cleartext Transmission of Sensitive Information[6]

Description

When connecting to the stacks node, the `auth_password` is transmitted in cleartext over HTTP connections (via Authorization header) due to the insecure protocol choice, exposing node credentials.

We can see that problem in this code:

```
        let send_request = || {
            Self.stacks_node_client
                .post(self.block_proposal_path())
                .header("Content-Type", "application/json")
                .header(AUTHORIZATION, self.auth_password.clone())
                .json(&block_proposal)
                .send()
                .map_err(backoff::Error::transient)
        };
```

The connection is established using plain HTTP, and this choice of protocol means that any data sent, including sensitive information like passwords, is transmitted in cleartext.
While it is stated in the documentation that this utility will run in an isolated, secure environment, the configuration file does not enforce this, and there is a risk that users will inadvertently send credentials through the network due to this issue.

Recommendation

Transition all communications to use HTTPS instead of HTTP. This will encrypt the data in transit, protecting sensitive information such as passwords from being intercepted. Alternatively, you could provide a default error message and confirmation if the user wants to send credentials through an unencrypted network instead of using a local interface like localhost.

[6] https://cwe.mitre.org/data/definitions/770.html

Status

Acknowledged. The Client is assumed to run in a trusted network environment and already include a warning message, however Coinfabrik still recommends printing an additional warning message when using another network address than localhost.

LO-05 Stale configuration Handling Of Reward Cycle

Location

- `./src/runloop.rs: 381`

Classification

- CWE-400: Uncontrolled Resource Consumption[7]

Description

The `refresh_runloop()` function in the Stacks Signer code reuses outdated reward cycle parameters, potentially leading to signers operating with stale configuration data after network upgrades.

The function `refresh_runloop()` is responsible for updating the reward cycle information, but it fails to fetch fresh values for `reward_cycle_length` and `prepare_phase_block_length` from the network. Instead, it reuses the existing values from the current `reward_cycle_info struct`. Any updates to those values will be ignored by the signer, causing operational failures.

Recommendation

Modify the `refresh_runloop()` function to always fetch the latest `reward_cycle_length` and `prepare_phase_block_length` from the network before updating the `reward_cycle_info`. This ensures that signers operate with the most current configuration.

Status

Acknowledged. `Reward_cycle_length/info` are hard-coded configuration values in the blockchain that never change, so they only need to be retrieved once.

[7] https://cwe.mitre.org/data/definitions/400.html

Enhancements

These items do not represent a security risk. They are best practices that we suggest implementing.

Id	Title	Status
EN-01	Exception Masking	Not implemented

EN-01 Exception Masking

Location

- `./src/http.rs: 239`

Description

The `retry_with_exponential_backoff` function maps all errors to `RetryTimeout`, discarding critical error context. In this code:

```
backoff::retry_notify(backoff_timer, request_fn, notify).map_err(|_| ClientError::RetryTimeout)
```

The `map_err` function converts permanent errors (e.g., invalid signatures) into a generic timeout, masking root causes. Callers cannot distinguish transient vs permanent failures, risking incorrect handling of security-critical errors.

Recommendation

Modify the `retry_with_exponential_backoff` function to categorize errors more effectively.

Status

Not implemented.

About CoinFabrik

CoinFabrik is a research and development company specialized in Web3, with a strong background in cybersecurity. Founded in 2014, we have worked on over 500 decentralization projects, including EVM-based and other platforms like Solana, Algorand, and Polkadot. Beyond development, we offer security audits through a dedicated in-house team of senior cybersecurity

professionals, working on code in languages such as Substrate, Solidity, Clarity, Rust, TEAL, and Stellar Soroban.

Our team has an academic background in computer science, software engineering, and mathematics, with accomplishments including academic publications, patents turned into products, and conference presentations. We actively research in collaboration with universities worldwide, such as Cornell, UCLA, and École Polytechnique in Paris, and maintain an ongoing collaboration on knowledge transfer and open-source projects with the University of Buenos Aires, Argentina. Our management and people experience team has extensive expertise in the field.

Methodology

CoinFabrik was provided with the source code, including automated tests that define the expected behavior, and general documentation about the project. Our auditors spent two weeks auditing the source code provided, which includes understanding the context of use, analyzing the boundaries of the expected behavior of each contract and function, understanding the implementation by the development team (including dependencies beyond the scope to be audited) and identifying possible situations in which the code allows the caller to reach a state that exposes some vulnerability. Without being limited to them, the audit process included the following analyses.

- Arithmetic errors
- Race conditions
- Misuse of block timestamps
- Denial of service attacks
- Missing or misused function qualifiers
- Needlessly complex code and contract interactions
- Poor or nonexistent error handling
- Insufficient validation of the input parameters
- Incorrect handling of cryptographic signatures
- Centralization and upgradeability

Severity Classification

Security risks are classified as follows[8]:

Critical	• Manipulation of governance voting result deviating from voted outcome and resulting in a direct change from intended effect of original results • Direct theft of any user funds, whether at-rest or in-motion, other than unclaimed yield • Direct theft of any user NFTs, whether at-rest or in-motion, other than unclaimed royalties • Permanent freezing of funds • Permanent freezing of NFTs • Unauthorized minting of NFTs • Predictable or manipulable RNG that results in abuse of the principal or NFT • Unintended alteration of what the NFT represents (e.g. token URI, payload, artistic content) • Protocol insolvency
High	• Theft of unclaimed yield • Theft of unclaimed royalties • Permanent freezing of unclaimed yield • Permanent freezing of unclaimed royalties • Temporary freezing of funds • Temporary freezing NFTs

[8] This classification is based on Immunefi severity classification system version 2.3. https://immunefi.com/immunefi-vulnerability-severity-classification-system-v2-3/

Medium	• Smart contract unable to operate due to lack of token funds • Block stuffing • Griefing (e.g. no profit motive for an attacker, but damage to the users or the protocol) • Theft of gas • Unbounded gas consumption • Security best practices not followed
Low	• Contract fails to deliver promised returns, but doesn't lose value • Other security issues with minor impact

Issue Status

An issue detected by this audit has one of the following statuses:

- **Unresolved:** The issue has not been resolved.
- **Resolved:** Adjusted program implementation to eliminate the risk.
- **Partially Resolved:** Adjusted program implementation to eliminate part of the risk. The other part remains in the code, but is a result of an intentional decision.
- **Acknowledged:** The issue remains in the code, but is a result of an intentional decision. The reported risk is accepted by the development team.
- **Mitigated:** Implemented actions to minimize the impact or likelihood of the risk.

Disclaimer

This audit report has been conducted on a **best-effort basis within a tight deadline defined by time and budget constraints**. We reviewed only the specific code provided by the client at the time of the audit, detailed in the Scope section. We do not review other components that are part of the solution: neither implementation, nor general design, nor business ideas that motivate them.

While we have employed the latest tools, techniques, and methodologies to identify potential vulnerabilities, **this report does not guarantee the absolute security of the contracts, as undiscovered vulnerabilities may still exist.** Our findings and recommendations are

suggestions to enhance security and functionality and are not obligations for the client to implement.

The results of this audit are valid solely for the code and configurations reviewed, and any modifications made after the audit are outside the scope of our responsibility. CoinFabrik disclaims all liability for any damages, losses, or legal consequences resulting from the use or misuse of the applications, including those arising from undiscovered vulnerabilities or changes made to the codebase after the audit.

This report is intended exclusively for the **Stacks** team and should not be relied upon by any third party without the explicit consent of CoinFabrik. Blockchain technology and smart contracts are inherently experimental and involve significant risk; users and investors should fully understand these risks before deploying or interacting with the audited contracts.

Changelog

Date	Description
10 April 2025	Initial report based on commit `c1a1f50fddcbc11054fae537103423e21221665a`.
18 April 2025	Fixes committed on `a7646f96e4e824e6c42ef0452df728b8618a01b2`.
30 April 2025	Final report based on commit `a7646f96e4e824e6c42ef0452df728b8618a01b2`