hashlock.

Security Audit

Sodax - Soroban (DeFi)

August 2025 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

The Sodax team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

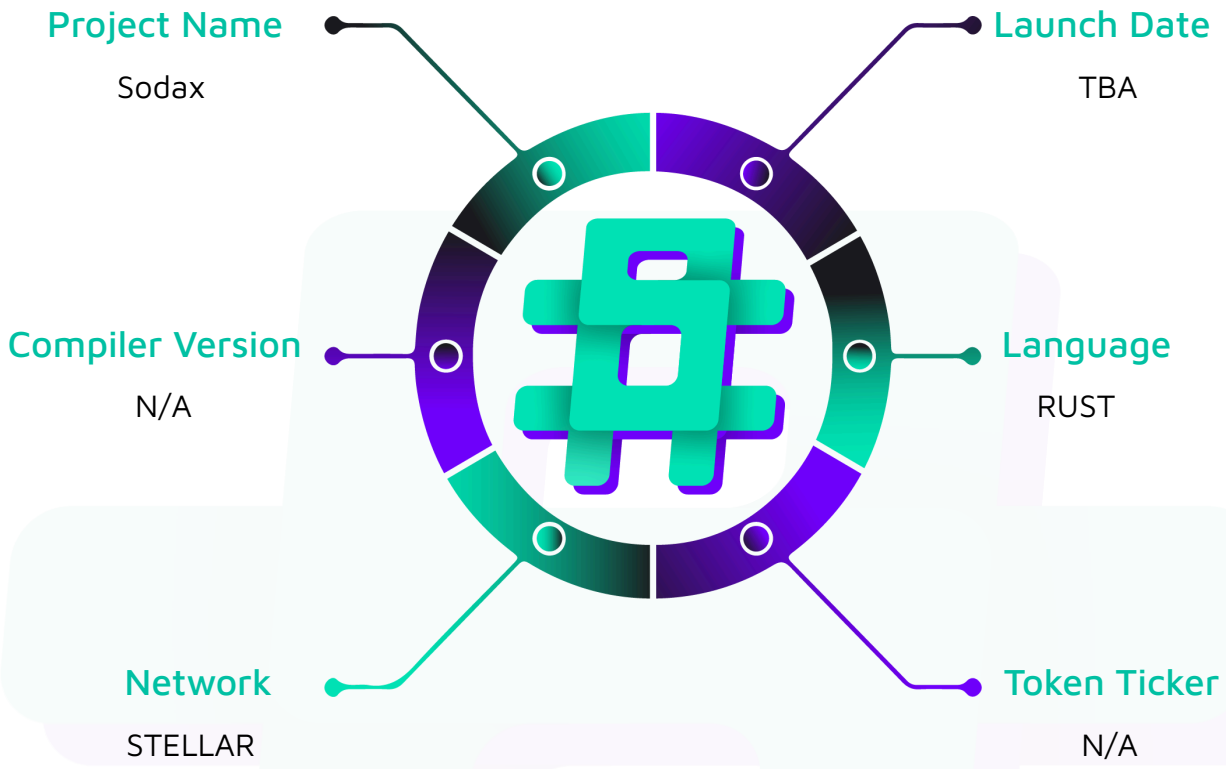
Project Context

The audit covers all core Soroban contracts built by the Sodax protocol, including:

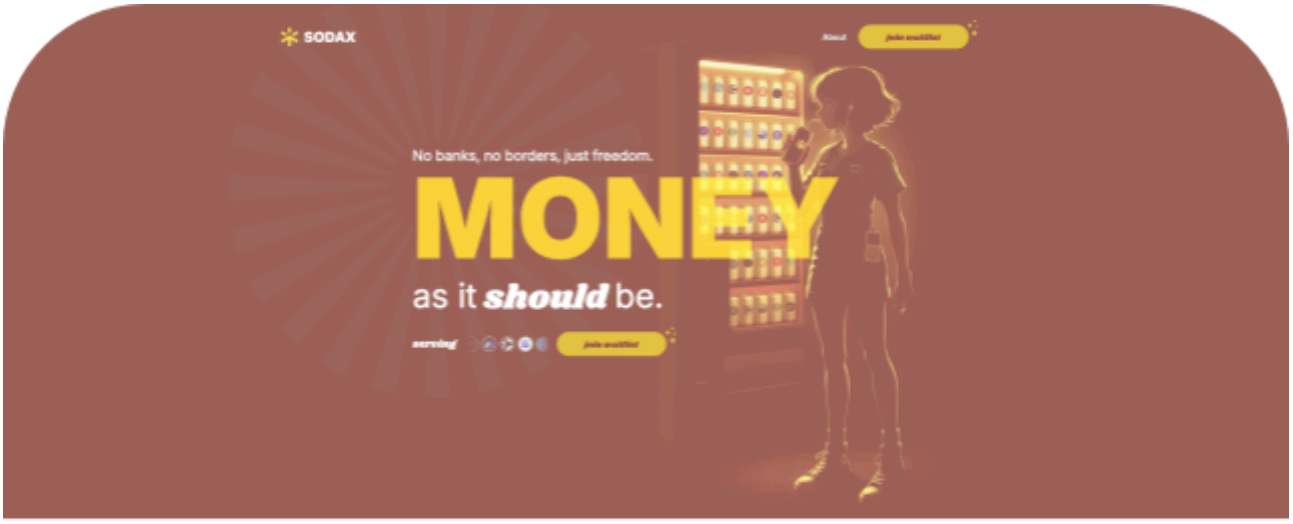
- Relay Connection Contracts – for managing cross-chain messaging and execution
- Asset Manager Contracts – for handling assets and execution flows
- Rate Limiter Contracts – for enforcing transaction and withdrawal limits

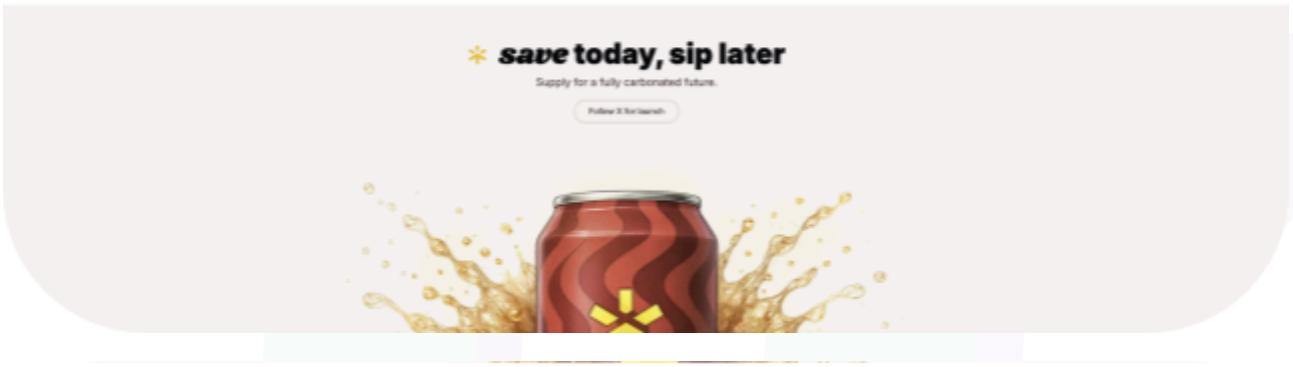
Project Name: Sodax
Project Type: DeFi
Website: https://www.sodax.com/
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the Soroban code within the Sodax project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	Sodax Smart Contracts
Platform	**Soroban / Rust**
Audit Date	**August, 2025**
Contract 1	- contracts/soroban/connectionv3/src/contract.rs - contracts/soroban/connectionv3/src/helpers.rs - contracts/soroban/connectionv3/src/event.rs - contracts/soroban/connectionv3/src/storage.rs
Contract 2	- contracts/asset-manager/src/contract.rs - contracts/asset-manager/src/storage.rs - contracts/asset-manager/src/helper.rs - contracts/asset-manager/src/types.rs
Contract 3	- contracts/rate-limit/src/contract.rs - contracts/rate-limit/src/storage.rs - contracts/rate-limit/src/helper.rs
Audited GitHub Commit Hash (1)	1ac3b7c78c1eadf6e17908ca216cd1e1026ef3f9
Audited GitHub Commit Hash (2)	3bd162435b57d4d734b2f68b9b0b614f0bfe78b8
Audited GitHub Commit Hash (3)	57eab5748002fbaec37303baeb6de42cb6d39cc1
Fix Review GitHub Commit Hash	3c8d5e3475610f1c55ae125979dfa3185684e90d

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

3 Medium severity vulnerabilities

5 Low severity vulnerabilities

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
AssetManager: – Allows users to: - Send tokens cross-chain - Submit inbound messages to withdraw - Allows admins to: - Rotate admin and configuration through set_admin, set_connection, set_rate_limit, and update hub chain via set_hub_info. - Upgrade contract code	**Contract achieves this functionality.**
RateLimit: - Allows users to: - Check and consume allowance for withdrawals - View contract state - Allows admins to: - Pause/unpause rate-limit checks. - Configure limits and infra through setting per-token limits and setting asset_manager, connection, and admin. - Upgrade contract code.	**Contract achieves this functionality.**
ConnectionV3: - Allows users to: - Send cross-chain messages - Verify incoming messages - Allows admins to: - Rotate admin, validator set, and threshold - Upgrade contract code	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Sodax project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Sodax project smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

Medium

[M-01] ConnectionV3#initialize - Setting zero threshold makes the bridge insecure

Description

ConnectionV3's `initialize` function sets the validator threshold to 0 and validators to an empty set, creating a window where signature verification has undefined behavior until validators are properly configured.

Vulnerability Details

The `initialize` function sets both validators and threshold to zero values:

```
pub fn initialize(env: Env, msg: InitializeMsg) -> Result<(), ContractError> {
        storage::is_initialized(&env)?;

        storage::store_chain_id(&env, msg.chain_id);
        storage::store_conn_sn(&env, 0);
        storage::store_admin(&env, msg.admin);
        storage::store_validator_threshold(&env, 0);
        storage::store_validators(&env, Vec::new(&env));

        Ok(())
    }
```

After initialization but before `update_validators` is called, the contract is in an inconsistent state. The `verify_signatures` function checks:

```
pub fn verify_signatures(
    e: &Env,
    signatures: Vec<BytesN<65>>,
```

```
    src_chain_id: &u128,
    src_address: &Bytes,
    conn_sn: &u128,
    message: &Bytes,
    dst_address: &Address,
) -> bool {
    let dst_network = storage::get_chain_id(e).unwrap();
    let validators = storage::get_validators(e).unwrap();
    let threshold = storage::get_validators_threshold(e).unwrap();

    if signatures.len() < threshold {
        return false;
    }
```

With the threshold set to 0, the function would return `true` even with no valid signatures. This creates a vulnerability window between contract deployment and validator configuration.

While `send_message` can be called immediately after deployment (only requiring `src_dapp_address.require_auth`), the real risk is that `verify_message` could theoretically pass verification without any valid signatures during this window.

Additionally, the `set_validators_threshold` function lacks minimum validation, allowing the threshold to be reset to 0 at any time:

```
pub fn set_validators_threshold(env: Env, threshold: u32) -> Result<(), ContractError> {
        helpers::ensure_admin(&env)?;
        let validators = storage::get_validators(&env).unwrap();
        if (validators.len() as u32) < threshold {
            return Err(ContractError::InsufficientValidators);
        }
        storage::store_validator_threshold(&env, threshold);
        Ok(())
    }
```

This inconsistency with `update_validators` (which requires threshold >= 1) means an admin could accidentally or maliciously set threshold to 0 after initialization, recreating the vulnerability. While the admin can fix this by setting a proper threshold later, any messages verified during the zero-threshold period would be accepted without proper authorization.

Impact

Temporary security vulnerability allowing message verification without signatures, potential for unauthorized bridge operations if exploited during the configuration window.

Recommendation

Either initialize with a non-zero threshold and at least one validator, or add validation to prevent operations until properly configured. Also, add minimum threshold validation to `set_validators_threshold` to match `update_validators`.

Status

Resolved

[M-02] soroban/contracts/asset-manager/src/contract.rs - Asset Loss via Fee-on-Transfer Tokens in Cross-Chain Transfers

Description

The asset-manager contract does not take into account the asset loss when a user transfers Fee-on-Transfer (FoT) tokens. The contract initiates a cross-chain message indicating the intended transfer amount, not the actual amount received after the fee is deducted.

```
pub fn transfer(
        env: Env,
        from: Address,
        token: Address,
        amount: u128,
        to: Bytes,
        data: Bytes,
    ) -> Result<(), ContractError> {
   // . . . existing code . . .
        let transfer_msg = TransferMessage::new(
            helper::address_to_bytes(&env, &token),
            helper::address_to_bytes(&env, &from),
            to,
            amount, // ← Vulnerability is using the user supplied amount
            data,
        );

        let transfer_msg_encoded = transfer_msg.encode(&env);
  // . . . existing code . . .
```

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

Impact

If the token has a transfer fee, currently uncommon on Stellar, but Soroban completely supports implementing a fee on the transfer of any token, and in that case, the assets-manager contract will receive fewer tokens than the amount.

However, the cross-chain message sent to the hub chain will state that the full amount was deposited. The hub chain will then credit the user with the full amount, creating assets that are not backed by corresponding funds on the spoke chain.

Recommendation

To fix this, the contract must not trust the user-provided amount. Instead, it must check its own balance of the token before and after the transfer to determine the actual amount received. This calculated, actual amount should then be used in the cross-chain message.

Status

Resolved

[M-03] soroban/contracts/rate-limit/src/helpers.rs#set_rate_limit&compute_available - Overestimation of Available Withdrawal Capacity Due To Asymmetric Balance Change

Description

The compute_available function overestimates withdrawal capacity by adding to available for detected increases (deposits) but not subtracting for decreases (e.g., external drains or outflows). This inflated value propagates to set_rate_limit, where it is computed and stored before baselines are updated, persisting the desynchronization and risking over-approvals in functions such as verify_withdraw.

Vulnerability Details

In compute_available, the logic adds refills and increases but ignores decreases:

```
let mut available = config.available + refill;

if current_balance > config.last_recorded_balance {

    available += current_balance - config.last_recorded_balance;

}

if available > config.max_available {

    available = config.max_available;

}
```

set_rate_limit calls this early, stores the potentially overestimated available, then updates baselines(e.g: last_recorded_balance)

Impact

Overestimation of the available rate could lead to approvals for withdrawals exceeding actual token holdings, potentially causing contract insolvency through over-drainage.

Recommendation

Consider adding symmetric subtraction in `compute_available` to underestimate capacity.

Status

Resolved

Low

[L-01] All contracts#set_admin - Single-step admin transfer allows accidental loss of contract control

Description

The current implementation across all three contracts (`AssetManager`, `RateLimit` and `ConnectionV3`) follows an identical pattern where admin privileges are transferred immediately without any validation or confirmation from the recipient address.

The issue is that `store_admin` immediately overwrites the existing admin address in storage without any validation that the new address is correct, accessible, or even intentionally provided. The Soroban `Address` type will accept any syntactically valid address, including addresses that may be valid on other chains but not on Stellar, addresses where the private keys are lost, or addresses resulting from typos in the input.

Once the admin is changed, there is no recovery mechanism. The old admin immediately loses all privileges and cannot revert the change. T

This is dangerous in a bridge context where admin functions control critical security parameters like validator sets, rate limits, and pause mechanisms.

Recommendation

Implement a two-step admin transfer process where the current admin proposes a new admin address and the new admin must claim ownership by calling `claim_admin`. This ensures the new address is valid and accessible before transferring control.

Status

Resolved

[L-02] RateLimit#set_rate_limit - Missing upper bound validation allows potential overflow scenarios

Description

The `rate_per_second` parameter in the `RateLimit` contract has no maximum value validation, allowing it to be set up to `u128::MAX`. While overflow is practically impossible with realistic time values, this could lead to unexpected behavior or effectively disable rate limiting.

The `set_rate_limit` function in `helper.rs` performs validation only on the lower bound of the `rate_per_second` parameter:

```
fn set_rate_limit(
    deps: DepsMut,
    env: Env,
    token: String,
    rate_per_second: u128,
    max_allowed_withdrawal: u128,
) -> Result<(), ContractError> {
    if rate_per_second == 0 {
        return Err(ContractError::InvalidRateLimit {});
    }

    let current_balance = get_balance_of(deps.as_ref(), &token)?;

    let config = RateLimitConfig {
        rate_per_second,
        max_available: max_allowed_withdrawal,
        last_updated: env.block.time.seconds(),
        last_recorded_balance: current_balance,
        available: 0,
    };

    save_rate_limit(deps.storage, &token, &config)?;
```

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

The absence of an upper bound check means administrators can set `rate_per_second` to extremely large values. When this value is used in the `compute_available` function, it's multiplied by `time_elapsed`:

While actual overflow would require unrealistic time periods (thousands of years), setting `rate_per_second` to very high values like `u128::MAX / 100` would effectively disable rate limiting since any reasonable `time_elapsed` would generate an available amount larger than any realistic withdrawal. This defeats the entire purpose of having rate limits as a security mechanism.

Recommendation

Add reasonable upper bound validation based on realistic token amounts and time periods.

Status

Acknowledged

[L-03] RateLimit#verify_withdraw - Unconfigured tokens bypass all rate limiting checks

Description

When a token has never been configured in the `RateLimit` contract, the `verify_withdraw` function immediately returns success, allowing unlimited withdrawals without any rate-limiting protection.

```
pub fn verify_withdraw(
        env: Env,
        _token: Address,
        _amount: u128,
        _balance: u128,
    ) -> Result<(), ContractError> {
        helper::ensure_asset_manager(&env)?;
        helper::ensure_not_paused(&env)?;



        let rate_limit_configs = storage::get_token_config(&env, _token.clone())?;
        if rate_limit_configs.rate_per_second == 0 {
            return Ok(());
        }
```

This design choice means that any ERC20-equivalent token on Stellar that hasn't been explicitly configured in the `RateLimit` contract can be withdrawn without any limits.

An attacker who gains control of the bridge or finds a vulnerability in the `AssetManager` could drain all unconfigured tokens instantly.

Recommendation

We recommend by default setting some specific rate limit for each new token added, to ensure that some, even high, limitation exists by design.

Status

Acknowledged

[L-04] ConnectionV3#verify_message - Out-of-order message processing breaks sequence guarantees

Description

ConnectionV3 allows messages to be verified in any order as long as the sequence number hasn't been used before. This breaks the ordering guarantees that cross-chain protocols typically depend on for state consistency.

The message verification logic in ConnectionV3's `verify_message` function only checks whether a specific sequence number has been previously used, without enforcing any ordering constraints.

```
pub fn verify_message(
        &self,
        src_chain_id: u128,
        src_address: Vec<u8>,
        conn_sn: u128,
        payload: Vec<u8>,
        signatures: Vec<Vec<u8>>,
    ) -> StdResult<SubMsg> {
        let wasm_msg = WasmMsg::Execute {
            contract_addr: self.connection.to_string(),
            msg: to_json_binary(&ExecuteMsg::VerifyMessage {
                src_chain_id,
                src_address,
                conn_sn,
                payload,
                signatures,
            })?,
            funds: vec![],
        };
        let cosm_msg = CosmosMsg::Wasm(wasm_msg);
        let submessage = SubMsg {
            id: CONNECTION_ID_PREFIX + 2,
```

```
            msg: cosm_msg,
            gas_limit: None,
            reply_on: cosmwasm_std::ReplyOn::Never,
            payload: vec![].into(),
        };
        Ok(submessage)
    }
```

The `get_sn_receipt` function simply returns a boolean indicating whether that specific sequence number from that specific source chain has been processed before. There's no tracking of what the next expected sequence number should be. This means if messages with sequence numbers [1, 2, 3, 4, 5] are sent from the source chain, they could be verified on the destination chain in any order, like [3, 1, 5, 2, 4].

This is problematic because many cross-chain protocols assume message ordering is preserved.

Recommendation

Track expected sequence number per source chain and enforce sequential processing to maintain message ordering guarantees.

Status

Acknowledged

[L-05] RateLimit#recv_message - Rate limit reset can be abused to bypass withdrawal restrictions

Description

The `reset_rate_limit` function immediately restores full withdrawal capacity without any cooldown period, allowing hub admin or signers to effectively bypass rate limits during an attack.

```
TransferType::ResetRateLimit => {
                helper::ensure_hub_admin_or_hub_signers(&env, &signer).unwrap();
                let decoded_reset_rate_limit =
                    msg::decode_reset_rate_limit(&env, message.data).unwrap();
                reset_rate_limit(&env, decoded_reset_rate_limit.token)?;
            }
```

The function immediately sets the `available` amount to `max_available`, completely restoring withdrawal capacity. This can be called through `recv_message` when processing cross-chain messages.

The authorization check allows either the hub admin or any single hub signer to execute this reset. This creates multiple problems. First, there's no cooldown period between resets - a compromised signer could reset limits repeatedly to allow continuous large withdrawals. Additionally, only one signer is needed rather than a threshold.

Recommendation

Implement a cooldown period between resets of at least 24 hours and require multiple signers or a higher threshold for reset operations.

Status

Acknowledged

Centralisation

The Sodax project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

Conclusion

After Hashlock's analysis, the Sodax project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au