hashlock.

Security Audit

Sodax - Sui (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.

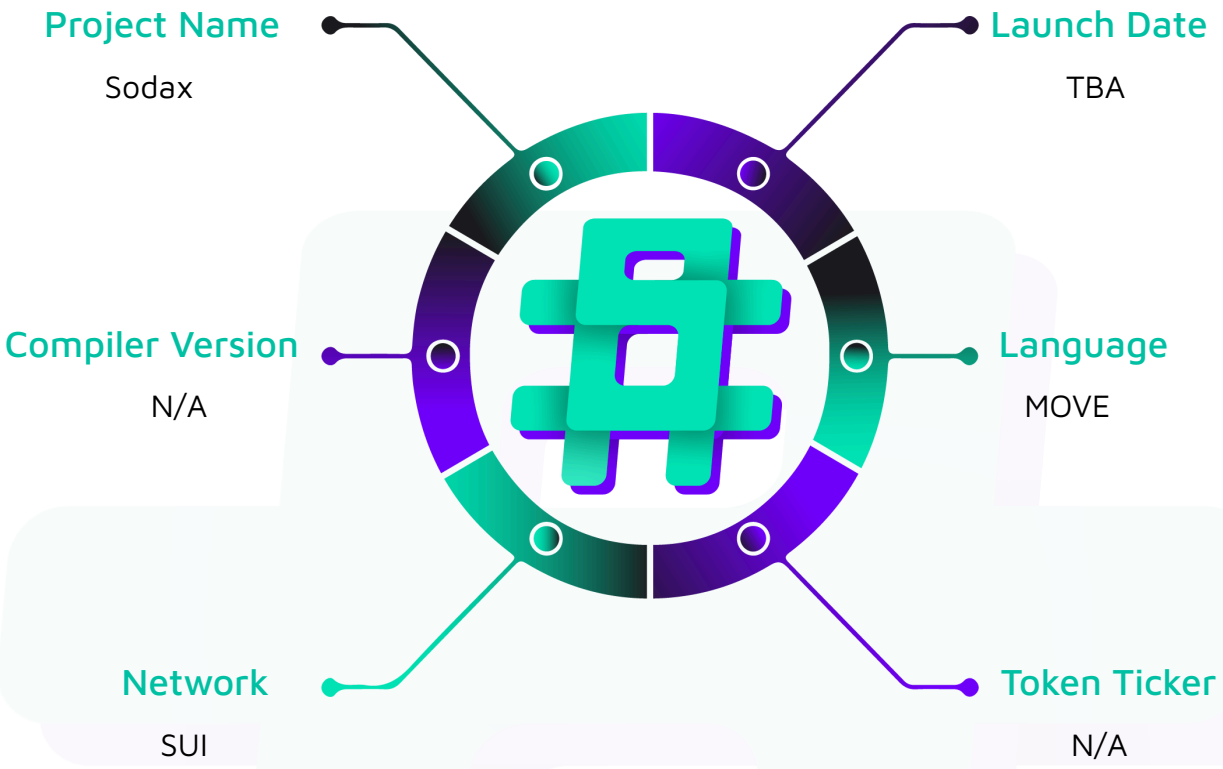
Project Context

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

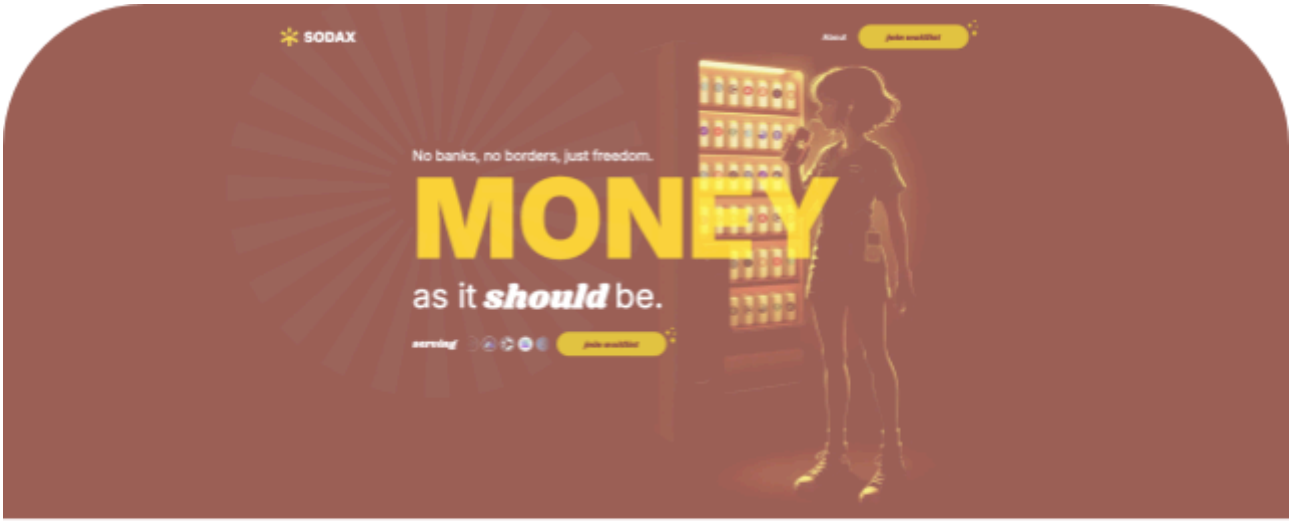
- Relay Connection Contracts – for managing cross-chain messaging and execution
- Asset Manager Contracts – for handling assets and execution flows
- Rate Limiter Contracts – for enforcing transaction and withdrawal limits

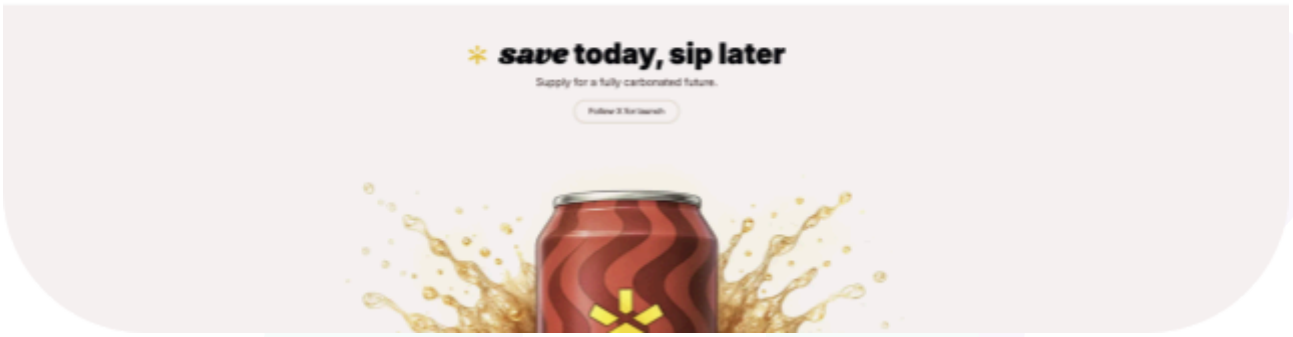
Project Name: Sodax
Project Type: DeFi
Website: https://www.sodax.com/
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the Move 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	**Sui / Move**
Audit Date	**August, 2025**
Component 1	- Connectionv3.move - utils.move
Component 2	- Asset_manager.move - Transfer_msg.move
Component 3	- rate_limits.move
Audited GitHub Commit Hash (1)	784fce34d90b39dc1a001bfab0e3832cf31d2321
Audited GitHub Commit Hash (2)	8de377ff8088f4d06fcbd8f2160df74dd0172072
Audited GitHub Commit Hash (3)	af46bfa02b5a3b0841603c52b9ff06bc18c0d9c8
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. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

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.

Hashlock found:

1 Medium severity vulnerability

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
connectionv3.move - Allows users to: - Send cross-chain messages to other networks - Verify incoming messages with validator signatures - Allows admins to: - Configure validator sets and thresholds - Initialize the system with the chain ID and validator configuration	**Contract achieves this functionality.**
utilis.move - Periphery contract used to: - Convert IDs/addresses to hex strings - Generate package addresses from publisher capabilities - Encode cross-chain message data for verification	**Contract achieves this functionality.**
asset_manager.move - Allows users to: - Transfer tokens cross-chain - Receive incoming cross-chain transfers - Allows admins to: - Set rate limit configurations - Configure hub network details	**Contract achieves this functionality.**
Transfer_msg.move - Create transfer messages with a token, sender, recipient, amount, and data - Encode and decode transfer messages	**Contract achieves this functionality.**

Rate_limits.move - Allows users to: - Verify withdrawal eligibility against rate limits - Query rate limit configurations and availability - Allows admins to: - Configure hub settings and asset manager - Pause/unpause the rate-limiting system - Receive cross-chain messages to update rate limits	**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] rate_limits.move - Missing ability to skip rate limiting

Description

The `rate_limits::rate_limits` module is designed to enforce per-token withdrawal rate limits, with an intentional bypass for tokens whose `rate_per_second` is set to `0`. In `verify_withdraw`, the function explicitly skips enforcement when `config.rate_per_second == 0`, which allows certain tokens to be exempt from rate limiting if configured as such.

However, the current implementation of `set_rate_limit_inner` prevents setting `rate_per_second` to `0` because of the assertion:

```
assert!(rate_per_second > 0, 0);
```

This directly blocks administrators from ever configuring a skip/no-limit mode for a token, after being set, even though `verify_withdraw` is built to support it.

Vulnerability Details

`verify_withdraw` logic:

```
if (config.rate_per_second == 0) return;
```

indicates the intended design: a zero `rate_per_second` should skip rate limit checks.

`set_rate_limit_inner` enforces:

```
assert!(rate_per_second > 0, 0);
```

This makes it impossible for `hub_admin` or authorized `hub_signers` to set `rate_per_second` to `0` via the `set_rate_limit` action.

Impact

Administrators cannot configure a token to bypass rate limits, even though the verification logic supports such a mode. This removes intended flexibility from the system.

Recommendation

Remove or adjust the assertion in `set_rate_limit_inner` to allow zero values:

```
// Current
assert!(rate_per_second > 0, 0);




// Recommended
assert!(rate_per_second >= 0, 0);
// or remove entirely if no other constraints are required
```

Status

Resolved

Low

[L-01] transfer_msg.move - Missing array bounds validation

Description

The `decode` function lacks validation of the decoded RLP list length, causing transaction aborts when processing malformed cross-chain messages

Vulnerability Details

The function assumes the decoded RLP list contains exactly 5 elements without validation

Proof of Concept

```
#[test]

fun test_decode_malformed_message() {

    // Create RLP with insufficient elements

    let mut incomplete_list = vector::empty<vector<u8>>();

    vector::push_back(&mut incomplete_list, encoder::encode(&b"token"));

    vector::push_back(&mut incomplete_list, encoder::encode(&b"from"));

    // Missing to, amount, data fields


    let malformed_bytes = encoder::encode_list(&incomplete_list, false);

    transfer_msg::decode(&malformed_bytes); // Aborts at vector::borrow index 2

}
```

Impact

- poor error handling
- wasted gas fees for legitimate transactions

Recommendation

Add length validation:

```
let decoded = decoder::decode_list(bytes);

assert!(vector::length(&decoded) == 5, E_INVALID_MESSAGE_FORMAT);
```

Status

Resolved

[L-02] rate_limits.move - Missing zero amount validation

Description

The `verify_withdraw` function lacks validation for zero-amount withdrawals

Vulnerability Details

The function processes withdrawals without checking if the amount is greater than zero

Proof of Concept

```
public fun verify_withdraw<T>(self: &mut Config, pub: &Publisher, balance: &sui::coin::Coin<T>, clock: &Clock, amount: u128) {
// lack of zero amount validation
    // ... other checks ...
    assert!(config.available >= amount, E_RATE_LIMIT_EXCEEDED);
    config.available = config.available - amount; // Zero subtraction allowed
}
```

Impact

Gas waste for users and potential spam attacks on the rate-limiting system through repeated zero-amount operations

Recommendation

Add amount validation:

```
assert!(amount > 0, E_INVALID_AMOUNT);
```

Status

Resolved

[L-03] rate_limits.move - Hard-coded error code instead of defined constant

Description

The `set_rate_limit_inner` function uses a hard-coded error code `0` instead of using a properly defined error constant for rate validation

Vulnerability Details

The function validates that `rate_per_second > 0`, but uses a hard-coded error code `0` in the assertion instead of defining and using a descriptive error constant like `E_INVALID_RATE_LIMIT`

Proof of Concept

```
fun set_rate_limit_inner(
    self: &mut Config,
    rate_per_second: u128,
    max_withdraw: u128,
    timestamp: u64,
    token: vector<u8>,
) {
    assert!(rate_per_second > 0, 0);  // Hard-coded error code
    // ... rest of function
}
```

Impact

Poor error handling and debugging experience due to non-descriptive error c

Recommendation

Define a proper error constant and use it in the assertion:

```
const E_INVALID_RATE_LIMIT: u64 = 10;

// Then use it in the function:

assert!(rate_per_second > 0, E_INVALID_RATE_LIMIT);
```

Status

Resolved

[L-04] rate_limits.move - Use of public_transfer instead of transfer for AdminCap

Description

Multiple modules use `public_transfer` instead of `transfer` when distributing `AdminCap` during initialization

Vulnerability Details

The `init` functions in `asset_manager.move`, `rate_limits.move`, and `connectionv3.move` use `public_transfer` for `AdminCap` distribution. From a security perspective, `transfer` should be used because it restricts transfers to only the defining module, following the principle of least privilege for sensitive administrative capabilities

Proof of Concept

```
// All three modules have this pattern:

fun init(otw: ASSET_MANAGER, ctx: &mut TxContext) {

    let admin_cap = AdminCap { id: object::new(ctx) };

    transfer::public_transfer(admin_cap, ctx.sender());

}
```

Impact

Violation of security best practices by not following the principle of least privilege for administrative capability transfers.

Recommendation

Use `transfer::transfer` instead of `transfer::public_transfer` for all `AdminCap` transfers:

```
transfer::transfer(admin_cap, ctx.sender());
```

Status

Resolved

[L-05] asset_manager.move - Unused error constant ENotOwner

Description

The contract declares an error constant `ENotOwner` that is never used

Vulnerability Details

The error constant is defined but never referenced

Proof of Concept

```
const ENotOwner: u64 = 0;  // Declared but never used

const ENotAssetManager: u64 = 1;  // Used in recv_message

const ENotUpgrade: u64 = 2;  // Used in migrate

const EWrongVersion: u64 = 3;  // Used in enforce_version
```

Impact

Code bloat and potential developer confusion about intended access control mechanisms

Recommendation

Either remove the unused constant or implement proper ownership checks

Status

Resolved

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. 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