hashlock.

Security Audit

Sodax - Solana (DeFi)

September 2025

info@hashlock.com.au

Hashlock Pty Ltd

Table of Contents

Executive Summary	4
Project Context	4
Audit Scope	7
Security Rating	9
Intended Smart Contract Functions	10
Code Quality	11
Audit Resources	11
Dependencies	11
Severity Definitions	12
Status Definitions	13
Audit Findings	14
Centralisation	32
Conclusion	33
Our Methodology	34
Disclaimers	36
About Hashlock	37

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.

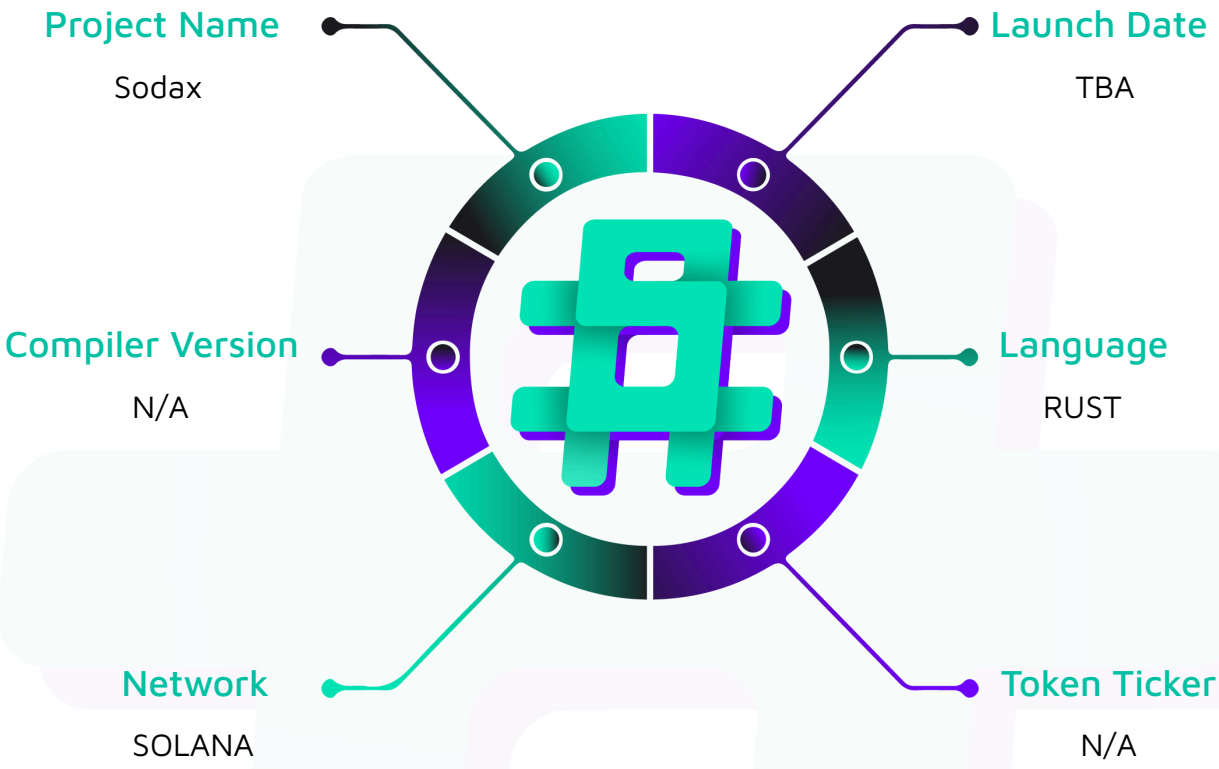
Project Context

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

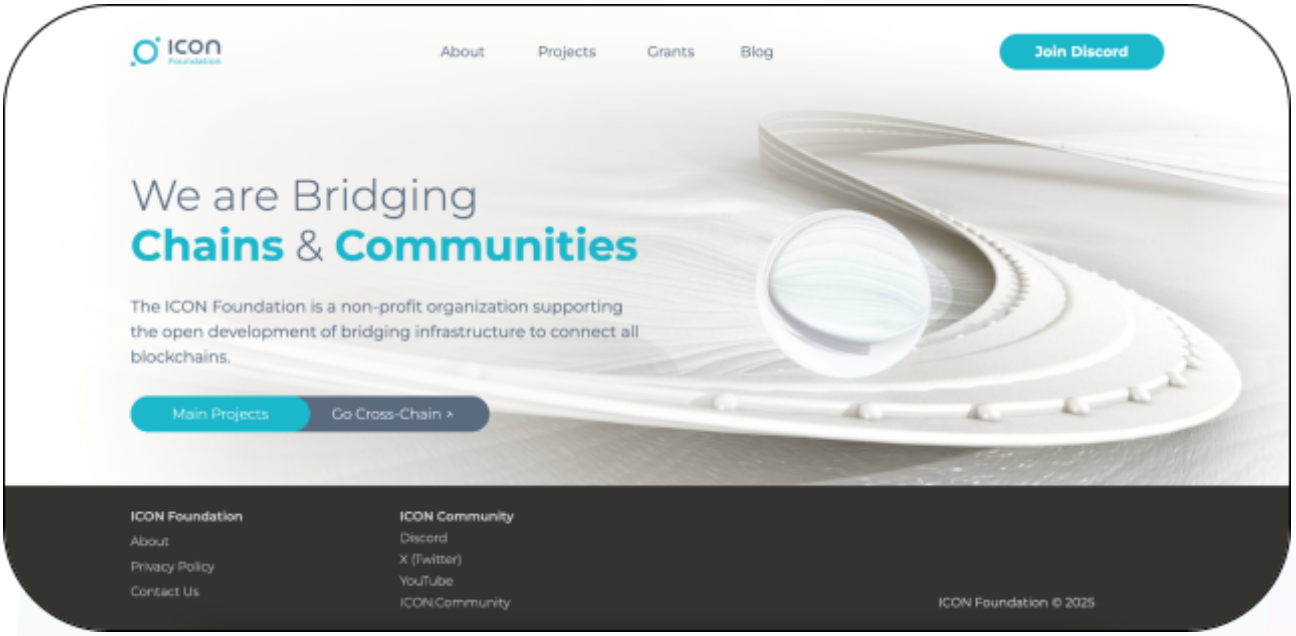
- Relay Connection Contracts – for managing cross-chain messaging and execution
- Asset Manager Contracts – for handling assets and execution flows
- Rate Limiter Contracts – for enforcing transaction and withdrawal limits

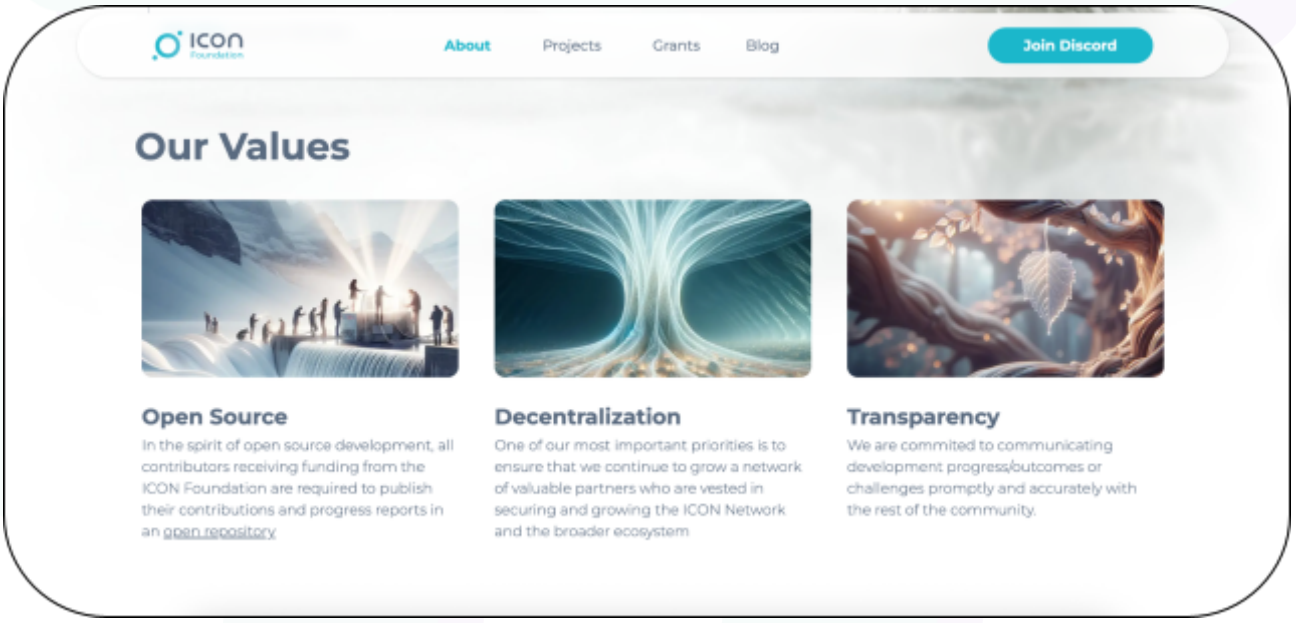
Project Name: Sodax
Project Type: DeFi
Website: https://www.sodax.com/
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the Solana 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	Solana / Rust
Audit Date	August, 2025
Contract 1	- contracts/solana/programs/connection/src/lib.rs -contracts/solana/programs/connection/src/query_accounts.rs - contracts/solana/programs/connection/src/state.rs - contracts/solana/programs/connection/src/helper.rs - contracts/solana/programs/connection/src/contexts.rs
Contract 2	- programs/asset-manager/src/lib.rs - programs/asset-manager/src/instructions/deposit.rs - programs/asset-manager/src/instructions/recv_msg.rs -programs/asset-manager/src/instructions/query_recv_msg_accounts.rs -programs/asset-manager/src/instructions/set_token_account_creation_fee.rs - programs/asset-manager/src/instructions/set_owner.rs - programs/asset-manager/src/instructions/initialize.rs - programs/asset-manager/src/helpers.rs - programs/asset-manager/src/states.rs - programs/asset-manager/src/structs
Contract 3	- programs/rate-limit/src/lib.rs - programs/rate-limit/src/errors.rs - programs/rate-limit/src/states.rs
Audited GitHub Commit Hash (1)	198cb79fe97c4c44390bee0eed3a950fa67f6766
Audited GitHub Commit Hash (2)	3c2cfd811ee36e423d5be90a4501fec72f3fd113
Audited GitHub Commit Hash (3)	8d53cca80586423b63fb5661c9e63bb967721593
Fix Review GitHub Commit Hash	80f8ed984825c8b02bbcee8dbf99ad09be9c0863

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 High severity vulnerabilities

2 Medium severity vulnerabilities

8 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.**
Connection: - 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

High

[H-01] RateLimit#verify_withdraw - Missing token scoping enables rate-limit bypass

Description

The `verify_withdraw` instruction accepts an arbitrary `RateLimitConfig` account and never ties it to the token passed in the instruction. It also early-exits as allowed when the referenced config's rate is zero. Together, this lets a caller point withdrawal checks at a different token's unthrottled config.

Vulnerability Details

The `VerifyWithdrawCtx` does not derive the `RateLimitConfig` PDA from the token being withdrawn, and the handler never asserts that the `rate_limit.token_mint == token` passed to the instruction. If an attacker supplies another token's `RateLimitConfig` that is configured with zero rate (treated as unlimited), the check returns success even though the target token's own limits would have blocked the withdrawal.

Because Anchor only enforces ownership on the passed account, any valid config for any mint can be substituted to "borrow" its looser policy, effectively nullifying per-mint throttling.

```
pub struct VerifyWithdrawCtx<'info> {
   #[account(mut)]
    pub signer: Signer<'info>,
    pub dapp: Signer<'info>,
    pub system_program: Program<'info, System>,

    #[account( mut )]
    pub rate_limit: Account<'info, RateLimitConfig>,
```

```
    #[account(
        seeds = [STATE_SEED],
        bump = config.bump
    )]
    pub config: Account<'info, State>,
}
```

Impact

Per-token rate limits can be bypassed, enabling withdrawals that exceed the intended throttle for a given mint.

Recommendation

Bind the `RateLimitConfig` account to the token via PDA seeds and also assert the config's stored mint equals the token argument. Consider removing the "zero means unlimited" behavior or gating it behind an explicit "disabled" flag that cannot be spoofed across tokens.

Status

Resolved

[H-02] AssetManager#recv_message - SPL mint not bound to the signed token address

Description

On the receive path, the program enforces rate-limit using the token address from the signed message, but actually spends and pays out using the SPL mint account supplied in transaction accounts. There is no assertion that these two mints are identical.

Vulnerability Details

In `asset_manager::instructions::recv_msg`, the RLP payload's `token_address` drives the native or SPL branch and is forwarded to the rate-limit CPI, but the SPL transfer path ultimately uses the mint account provided in `ctx.accounts` and derives vault PDAs from `admin_token_account.mint`.

There is no assertion that `mint.key() == token_address` from the signed message. An attacker can submit a signed message for mint X while supplying accounts for mint Y.

The program will enforce rate-limit on X, but spend from the Y vault and send Y to the recipient, effectively draining Y under X's limits.

```
pub fn recv_message<'info>(
    ctx: Context<'_, '_, '_, 'info, ReceiveMessageCtx<'info>>,
    src_chain_id: u128,
    src_address: Vec<u8>,
    conn_sn: u128,
    payload: Vec<u8>,
    signatures: Vec<[u8; 65]>,
) -> Result<()> {
    let config = &ctx.accounts.config;
    require_eq!(
        config.hub_chain_id,
        src_chain_id,
        AssetManagerError::InvalidHubNid
    );
```

```
    require!(
        src_address == config.hub_asset_manager,
        AssetManagerError::InvalidAssetManager
    );


    let rlp_data: &rlp::Rlp<'_> = &rlp::Rlp::new(&payload);
    let transfer_details = TransferData::decode(rlp_data).expect("decode error");


    let token_address = Pubkey::try_from(transfer_details.token_address.as_slice())
        .map_err(|_| error!(AssetManagerError::InvalidTokenAddress))?;
    let native_token = Pubkey::default();


    let recipient = Pubkey::try_from(transfer_details.to.as_slice())
        .map_err(|_| error!(AssetManagerError::InvalidRecipientAddress))?;
    let to_native = &ctx.accounts.to_native;
```

Impact

Asset mix-up and potential draining of one mint under another mint's limits; cross-asset accounting divergence.

Recommendation

Before any SPL operations, require the passed SPL mint to equal the token address from the verified message, and derive the vault strictly from that verified mint.

Status

Resolved

Medium

[M-01] programs/asset-manager/src/instructions/initialize.rs - Program initialization vulnerable to frontrun

Description

The `initialize` function allows any signer to call it and set themselves as the owner without restrictions, using `init_if_needed` with fixed seeds. This enables frontrunning, where a malicious actor calls it first after deployment, claiming ownership and setting arbitrary parameters.

Impact

The program could be hijacked or misconfigured, rendering it unusable, requiring redeployment, costing time, and SOL.

Recommendation

Add validation to restrict calls to an expected deployer key.

Status

Resolved

[M-02] 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 new(chain_id: u128, admin: Pubkey, bump: u8) -> Self {
        Self {
            admin,
            chain_id,
            validators: Vec::new(),
            threshold: 0,
            sn: 0,
            bump,
        }
    }
```

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

```
if (unique_validators.len() as u8) < ctx.accounts.config.threshold {
        return Err(ConnectionError::ValidatorsMustBeGreaterThanThreshold.into());
    }
    Ok(())
```

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, the real risk is that `verify_message` could theoretically pass verification without any valid signatures during this window.

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

```
pub fn set_threshold(ctx: Context<SetConfigItem>, threshold: u8) -> Result<()> {
        if ctx.accounts.config.validators.len() < threshold as usize {
                                                                    return
Err(error::ConnectionError::ValidatorsMustBeGreaterThanThreshold.into());
        }
        ctx.accounts.config.threshold = threshold;
        Ok(())
    }
```

This means that an admin could accidentally or maliciously set a 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.

It is worth mentioning that, for example, the Soroban side of the bridge enforces that `update_validators` require a threshold to be at least 1.

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`.

Status

Resolved

Low

[L-01] programs/rate-limit/src/states.rs - Incorrect account space allocation in the State struct

Description

The `SPACE` constant for the account underestimates the required serialized size by 88 bytes, as it fails to account for the 4-byte (u32) length prefixes in serialization for `Vec<u8>` fields (`hub_admin`, `hub_manager`, and each inner `Vec` `hub_signers`).

Recommendation

Update the calculation to include length prefixes for all Vec fields:

```
pub const SPACE: usize = 8 + // discriminator

    1 + // paused

    32 + // asset_manager

    32 + // connection

    4 + 20 + // hub_admin (len + max elements)

    4 + 20 + // hub_manager

    16 + // hub_chain_id

     4 + (MAX_HUB_SIGNERS * (4 + 20)) + // hub_signers (outer len + inners: len + max elements each)

    32 + // authority

    1; // bump
```

Status

Resolved

[L-02] AssetManager#recv_message - Hardcoded rent constant leads to fee mischarging

Description

Inside `instructions::recv_message::handler` (SPL branch), the code treats the recipient's account as rent-exempt if `lamports ≥ 2_039_280`, then conditionally charges a per-mint fee and creates the ATA. This mixes an outdated "magic number" with dynamic rent rules and account sizes, so the check can drift from reality over time.

Because the fee is deducted before the ATA creation call and hinges on that fixed constant rather than querying cluster rent or detecting ATA existence, users can be over or under charged, and behavior may diverge across clusters or upgrades.

Recommendation

Use the rent sysvar to compute the current minimum, or gate fee collection on whether an ATA was actually created (idempotent ATA create) rather than on a lamports heuristic.

Status

Resolved

[L-03] RateLimit - Zeroed rate per second treated inconsistently across read and verify paths

Description

The `instructions::verify_withdraw::handler` returns success immediately when `rate_per_second == 0`, effectively disabling enforcement for that token, while `get_available` returns `0` for the same state, implying no available capacity to observers.

The split behavior is visible in the same module. This inconsistency means operators and dashboards can believe withdrawals are blocked when they are in fact unrestricted.

Recommendation

Pick a single semantic, for example, zero = disabled or zero = unlimited, implement it consistently in both functions, and document it clearly.

Status

Acknowledged

[L-04] RateLimit#recv_message - Unbounded message validity window

Description

The `instructions::recv_message::handler` only checks `message.deadline ≥ now`, meaning a message signed today with a far-future deadline remains valid until that time. The `deadline` field is parsed from the RLP-encoded admin `Message` and not otherwise bound.

If a signer's key is compromised, attackers can keep using these long-lived admin messages for a long time, which makes it harder to stop them and recover.

Recommendation

Enforce a maximum TTL (e.g., hours/days) by requiring a `deadline - now ≤ configured cap`. Additionally, reject messages with excessively long validity.

Status

Acknowledged

[L-05] AssetManager - No guard against mints with active freeze authority

Description

In the SPL branch of `instructions::recv_message::handler`, the code loads the `mint` account and proceeds to ATA checks and token transfers without asserting `mint.freeze_authority.is_none`.

If the mint retains a freeze authority, a privileged entity can freeze vault or recipient accounts post-deposit, halting redemptions while the bridge appears healthy, creating an unexpected liveness dependency.

Recommendation

Either enforce `freeze_authority == None` for allowed mints, or explicitly accept the risk and document it properly. If allowed, consider allow-listing and periodic checks.

Status

Resolved

[L-06] RateLimit - Important configuration changes lack events

Description

The `instructions::admin_actions::set_asset_manager` and `set_connection` mutate the primary trust anchors for verification and dApp authority without emitting indexed events, unlike pause, unpause, and signer updates, which do emit.

Without events, off-chain monitoring and audit trails can miss or delay detection of these sensitive updates.

Recommendation

Emit clear, indexed events for these updates.

Status

Acknowledged

[L-07] RateLimit#actions::SetRateLimit - Missing upper bound validation allows potential overflow scenarios

Description

The `rate_per_second` parameter in `RateLimit` program has no maximum value validation, allowing it to be set up to `u64::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:

```
pub fn set_rate_limit(
    rate_limit: &mut RateLimitConfig,
    mint: Pubkey,
    current_time: i64,
    rate_per_second: u64,
    max_available: u64,
) -> Result<()> {
    require_gt!(rate_per_second, 0, RateLimitError::InvalidRateLimit);
    require_gt!(max_available, 0, RateLimitError::InvalidRateLimit);



    rate_limit.rate_per_second = rate_per_second;
    rate_limit.max_available = max_available;
    rate_limit.last_updated = current_time;
    rate_limit.last_recorded_balance = 0;
    rate_limit.available = 0;
    rate_limit.is_initialized = true;
    rate_limit.token_mint = mint;



    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 `u64::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-08] 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.

```
Actions::ResetRateLimit { token } => {
            verify_hub_admin_or_signers(&state, &sender.to_vec())?;



            let rate_limit_account = ctx
            .accounts
            .rate_limit
            .as_ref()
            .ok_or(RateLimitError::InvalidRateLimitAccount)?;
            let (expected_rate_limit, _) = Pubkey::find_program_address(
                &[RATE_LIMIT_SEED, token.as_ref()],
                ctx.program_id,
            );
            require_keys_eq!(
                rate_limit_account.key(),
                expected_rate_limit,
                RateLimitError::InvalidTokenAccount
            );
            let mut rate_limit_data = rate_limit_account.try_borrow_mut_data()?;
                            let mut rate_limit = RateLimitConfig::try_deserialize(&mut
rate_limit_data.as_ref())?;


            reset_rate_limit(&mut rate_limit, current_time)?;
            let new_available = rate_limit.available;
```

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

Resolved

[L-09] programs/rate-limit/src/helper.rs#recover_sender - Signature Malleability

Description

The program uses `secp256k1` ECDSA signatures to authenticate privileged messages in a cross chain environment. However, it does not enforce low-`s` value checks (signature malleability protection) on these signatures, allowing attackers to submit malleable variants of valid signatures by flipping the `s` value.

Recommendation

Enforce low-`s` signature checks (reject signatures with s > n/2; as recommended in the seckp256k1 crate doc) during signature verification, to eliminate malleability risks.

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 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