# Security Audit

## Energy Web X Liquid Staking Pallet (DeFi)

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

## Table of Contents

CAUTION

## Executive Summary

The Energy Web Foundation 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

Energy Web is a global, open-source nonprofit focused on accelerating the clean energy transition through decentralized digital infrastructure. Launched in 2017, the organization stewarded the Energy Web Chain (EWC), an enterprise-focused Proof-of-Authority blockchain. Energy Web is now transitioning to it's flagship network: Energy Web X (EWX), a Substrate-based Polkadot parachain.

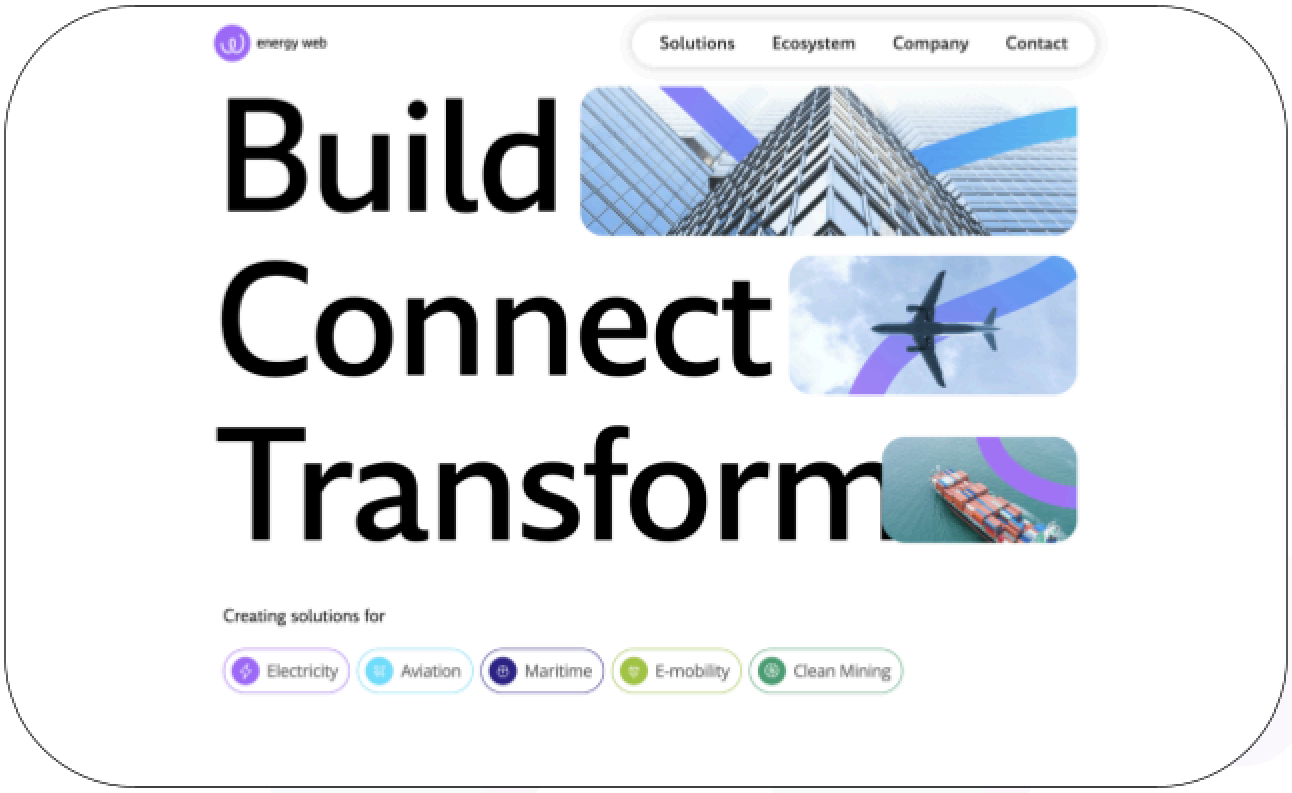
EWX introduces a permissionless Proof-of-Stake consensus model, enabling broad validator and delegator participation while unlocking staking rewards for participants and supporting a robust on-chain economy. To expand liquidity and interoperability, the Energy Web Token (EWT) is transitioning into a fully compliant ERC-20 token on Ethereum mainnet, supported by a dual bridge architecture: a bidirectional bridge between Ethereum and EWX as well as continued support for lifting from EWC to EWX. trading, advancing interoperability, regulatory compliance, and grid decentralization.

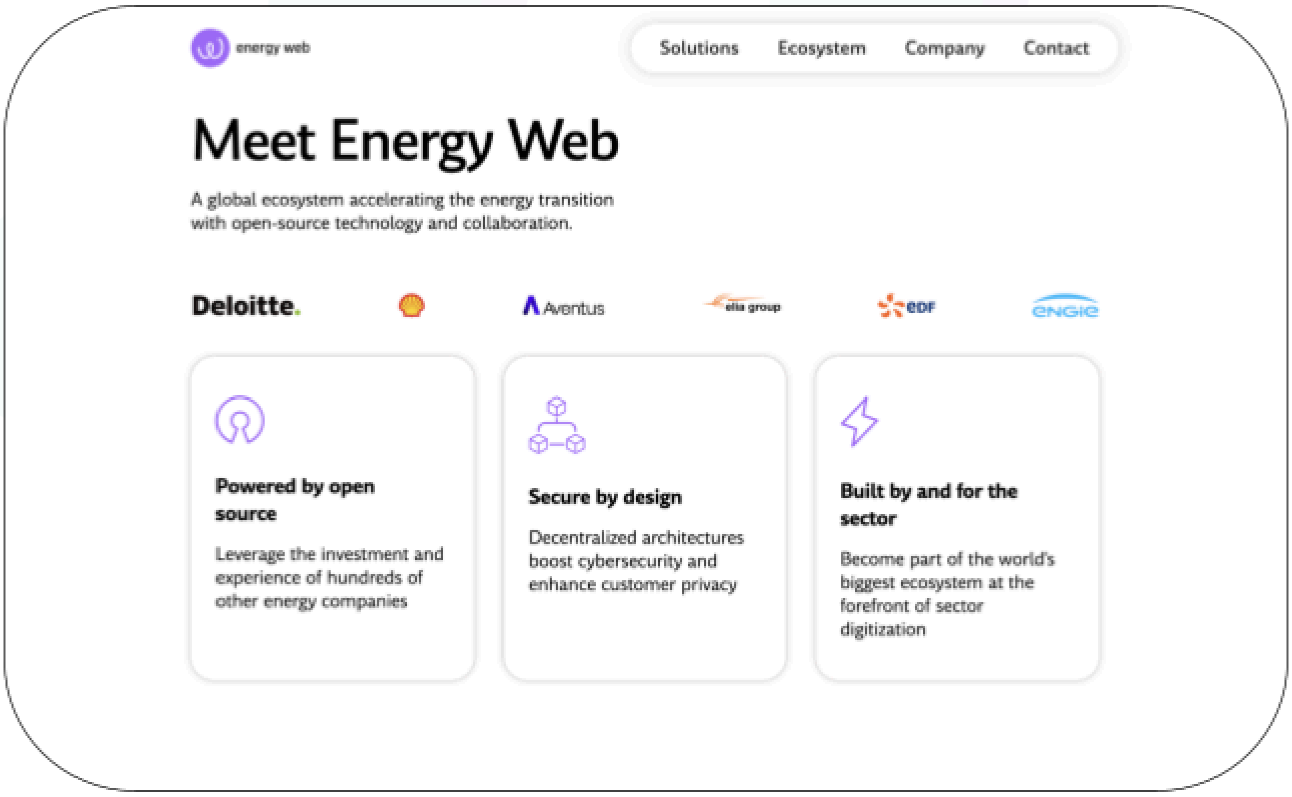
**Project Name**: Energy Web X Liquid Staking
**Project Type:** DeFi, Token, Bridge
**Website:** https://www.energyweb.org/
**Logo:**

**Visualised Context:**

| | |
|---|---|
| Project Name | Energy Web X Liquid Staking |
| Launch Date | 15/10/2025 |
| Compiler Version | N/A |
| Language | RUST |
| Network | POLKADOT |
| Token Ticker | EWT |

**Project Visuals:**




Meet Energy Web

A global ecosystem accelerating the energy transition with open-source technology and collaboration.

**Powered by open source**

Leverage the investment and experience of hundreds of other energy companies

**Secure by design**

Decentralized architectures boost cybersecurity and enhance customer privacy

**Built by and for the sector**

Become part of the world's biggest ecosystem at the forefront of sector digitization

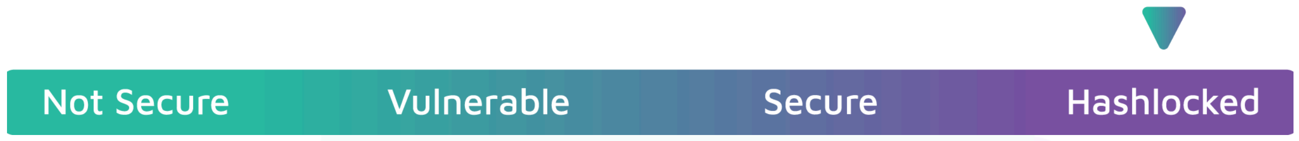
## Audit Scope

We at Hashlock audited the pallet code within the Energy Web X Liquid Staking project. The scope of work included a comprehensive review of the pallet components listed below. We tested the pallet 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** | **Energy Web X Liquid Staking** |
|---|---|
| **Platform** | **Polkadot / Rust** |
| **Audit Date** | **August, 2025** |
| **Scope** | https://github.com/energywebfoundation/energy-web-parachain-node/pull/202 |
| **Audited GitHub Commit Hash** | 5ee29af4cbafd37e60b76988d49a13c6ecc2e654 |
| **Fix Review GitHub Commit Hash** | 34a9c849d51a1a0d3e5e11cc20bdc133d2979061 |

## Security Rating

After Hashlock's Audit, we found the pallet to be **"Hashlocked"**. The pallet code 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 pallet functionality is presented in the Intended Pallet Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

**Hashlock found:**

2 Medium severity vulnerabilities

3 Low severity vulnerabilities

1 QA

**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 Pallet Functions

| Claimed Behaviour | Actual Behaviour |
|---|---|
| **pallet-liquid-staking**<br><br>- Stake native tokens to mint voucher tokens<br>- Burn vouchers to request unstake and enqueue<br>- Batch and schedule unbonding to selected collators<br>- Create per-user unlock chunks with claim eras<br>- Execute matured unbonds and fund the Unstaking Pot<br>- Users claim native tokens from the Unstaking Pot<br>- Auto-restake rewards and idle balance to compound<br>- Manage a whitelist of collators with priority drains<br>- Allow administrator staking top-ups via StakeAdmin | **Pallet achieves this functionality.** |

## Code Quality

This audit scope involves the liquid staking pallet of the Energy Web X Liquid Staking project, as outlined in the Audit Scope section. All components 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 Energy Web X Liquid Staking project pallet 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 dependencies used in this parachain infrastructure are based on well-known industry standard open source projects.

## 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] pallet-liquid-staking - Special accounts can be reaped through AllowDeath

#### Description

Transfers into and out of the unstaking pot and from the pallet account use `ExistenceRequirement::AllowDeath`. If either the unstake pot or the pallet account gets reaped, subsequent transfers that are below the existential deposit to recreate the account will fail, and the claim and unlock pipeline can get stuck.

#### Vulnerability Details

Transfers that move funds between the pallet account returned by `account_id` and the `UnstakingPotId` use `Balances::transfer` with `ExistenceRequirement::AllowDeath`. If either of these system accounts gets reaped, then any subsequent pallet to pot credit that is below the existential deposit will fail to create the destination and return an error, which causes the surrounding `with_transaction` in `on_initialize` to roll back the entire phase.

```
// Transfer claimed funds to the dedicated unstaking pot.
                if T::Currency::transfer(
                    pallet_account,
                    &unstake_pot_account,
                    unlock_chunk.amount,
                    ExistenceRequirement::AllowDeath,
                )
```

It is worth mentioning that this issue occurs in three functions:

- `claim_unstake`
- `do_update_claim_unbonded_funds_storage`

- `do_update_try_ensure_balance_is_reinvestable_storage`

### Impact

Claim funding can stall for many blocks, which translates to user payouts not progressing even though unlocks have matured.

### Recommendation

We recommend replacing `AllowDeath` with `KeepAlive` for all transfers involving the pallet account and `UnstakingPotId`. Seed both with at least the existential deposit and, optionally, accumulate funding transfers until they are greater than or equal to the existential deposit before attempting the transfer.

### Status

Resolved

## [M-02] pallet-liquid-staking#try_process_unbond_requests - Unbounded unstake queue might lead to Denial of Service attack

### Description

`QueuedUnstakeRequests` has no per-system bound and is explicitly unbounded by design. There's no storage deposit per entry to disincentivize spam - the "protection" is only `MinUnstakingAmount`.

A potential attacker can create many funded accounts, stake once, then queue minimum-sized unstakes, permanently bloating the on-chain state for free.

### Vulnerability Details

The `QueuedUnstakeRequests` is intentionally unbounded, and there is no storage deposit per entry. A potential attacker can create many minimally funded accounts, then call `request_unstake` with `MinUnstakingAmount` repeatedly to leave permanent entries that the chain must scan and maintain.

```
// Transfer claimed funds to the dedicated unstaking pot.
                if T::Currency::transfer(
                    pallet_account,
                    &unstake_pot_account,
                    unlock_chunk.amount,
                    ExistenceRequirement::AllowDeath,
                )
```

### Impact

Long-term state bloat and heavier per-iteration, which degrade performance and increase withdrawal latency under load.

### Recommendation

We recommend requiring a refundable storage deposit per queued request, or enforcing per origin and global bounds, and increasing `MinUnstakingAmount` to at least cover the on-chain storage cost of an entry.

**Status**

Resolved

## Low

### [L-01] pallet-liquid-staking - Documentation overstates spam protection without deposits

#### Description

Documentation claims `MinUnstakingAmount` makes spam economically unfeasible, but the code does not require a storage deposit, and the queue is unbounded. The threat model is therefore optimistic if the token price or fees change.

```
pub const MinUnstakingAmount: Balance = EWT;
```

As currently EWT token is priced as +- $1, such protection might not be working as expected.

#### Recommendation

We recommend updating the operational guidance to reflect the real risk and add storage deposits and or hard bounds so economics align with security.

#### Status

Resolved

## [L-02] pallet-liquid-staking - Collator slot griefing through dust requests

### Description

Collator slot griefing is possible because the pallet can schedule at most one pending request per nominator to the collator pair, and the `ProcessUnbondRequests` phase touches one collator per block.

An attacker that meets `MinUnstakingAmount` can repeatedly occupy the single slot on each whitelisted collator with dus,t which forces larger unstakes to wait full unbonding windows.

```
let transaction_result = with_transaction(|| match current_phase {
                ProcessingPhase::ProcessUnbondRequests => {
                    let iterations = {
                        match max_iterations {
                            Some(max) if max > 0 => max,
                            _ => {
                                                                                            return
TransactionOutcome::Rollback(Err(DispatchError::Other(
                                    "Failed to process unbond requests",
                                )))
                            },
                        }
                    };
```

### Recommendation

We recommend adding per-origin rate limits or cool-downs for `request_unstake`. Enforce a per collator minimum batch for scheduling and prioritize larger and older requests to prevent dust from preempting meaningful work. Require a storage deposit per queued request to raise the attack cost.

### Status

Acknowledged

## [L-03] pallet-liquid-staking#can_insert_any_unlock_chunk - Parameter missconfig can brick processing

**Description**

The `can_insert_any_unlock_chunk` gates admission using `T::MaxUnlockChunks::get().saturating_sub(T::MaxWhitelistedCollators::get(),` which only prevents new requests when headroom collapses to zero.

Once requests are already queued, the write path in `do_update_user_unbond_storage` still needs to append an `UnlockChunk` into a `BoundedVec` sized by the current `MaxUnlockChunks`. If governance shrinks `MaxUnlockChunks` or effectively raises the worst case scenario by growing `MaxWhitelistedCollators` after admission, then `chunks.try_push` can return an error, which causes the surrounding `with_transaction` to roll back the whole phase.

A related processing time failure exists in the per-era bookkeeping, where `PendingUnstakeEra` is placed into a `BoundedBTreeMap` bounded by `MaxWhitelistedCollators`. Conversions or inserts such as `try_from` or `try_insert` can fail once the number of collators to write exceeds the new bound, which triggers the same rollback.

The `saturating_sub` clamp at admission does not defend these processing time writes so the scenario remains exploitable.

```
pub fn can_insert_any_unlock_chunk(who: &T::AccountId) -> bool {
            let pending_unstake_chunks = ClaimableAmount::<T>::decode_len(who);
            let max_safe_unlock_chunks = (T::MaxUnlockChunks::get() as usize)
                .saturating_sub(T::MaxWhitelistedCollators::get() as usize);
            if let Some(pending_unstake_chunks) = pending_unstake_chunks {
                pending_unstake_chunks <= max_safe_unlock_chunks
            } else {
                true
            }
        }
```

Repeated rollbacks stall the state machine and block both scheduling and claim funding, which disallows users from unlocking until parameters are reverted or the state is manually repaired.

**Recommendation**

We recommend revalidating capacity at processing time before `try_push` or `try_insert` and deferring excess work rather than returning an error that rolls back the transaction.

**Status**

Acknowledged

# QA

### [Q-01] pallet-liquid-staking#set_whitelisted_collators - Whitelist allows duplicate collators

#### Description

The whitelist accepts duplicates because `set_whitelisted_collators` writes a `BoundedVec` without deduplication, and selection helpers like `get_next_unstake_collator` assume a set. Duplicates bias ordering and can cause repeated work on the same collator, which wastes block weight.

#### Recommendation

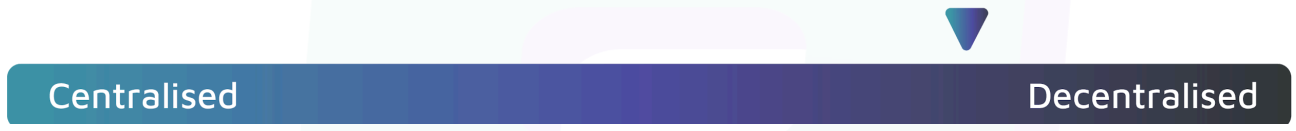
We recommend enforcing setting semantics on input by rejecting lists with duplicates or deduplicating internally before writing to storage.

#### Status

Resolved

## Centralisation

The Energy Web X Liquid Staking project is moving toward full decentralization by having many independent validators make all key decisions instead of a single team. A temporary admin role is only in place during upgrades, after which governance will be fully community-driven.



## Conclusion

After Hashlock's analysis, the Energy Web X Liquid Staking 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 components 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 pallet details are made public.

# Disclaimers

## Hashlock’s Disclaimer

Hashlock’s team has analysed these components in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the pallet 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 pallet. 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 pallet

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

Pallets are executed on a blockchain platform. The platform, its programming language, and other software related to the parachains can have their own vulnerabilities that can lead to attacks. Thus, the audit can’t guarantee the explicit security of the audited codebase.

## 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