Security Audit

Energy Web (Blockchain)

August 2025

info@hashlock.com.au

Hashlock Pty Ltd

Table of Contents

Executive Summary 4
Project Context 4
Audit Scope 7
Security Rating 8
Intended Parachain Functions 9
Code Quality 12
Audit Resources 12
Dependencies 12
Severity Definitions 13
Status Definitions 14
Audit Findings 14
Centralisation 36
Conclusion 37
Our Methodology 38
Disclaimers 40
About Hashlock 41

CAUTION

Executive Summary

The Energy Web team partnered with Hashlock to conduct a security audit of their parachains. 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: The Energy Web Foundation
Project Type: DeFi, Token, Bridge
Website: https://www.energyweb.org/
Logo:

Visualised Context:

Project Name	Energy Web
Launch Date	11/09/2025
Compiler Version	N/A
Language	RUST
Network	POLKADOT
Token Ticker	N/A

Project Visuals:

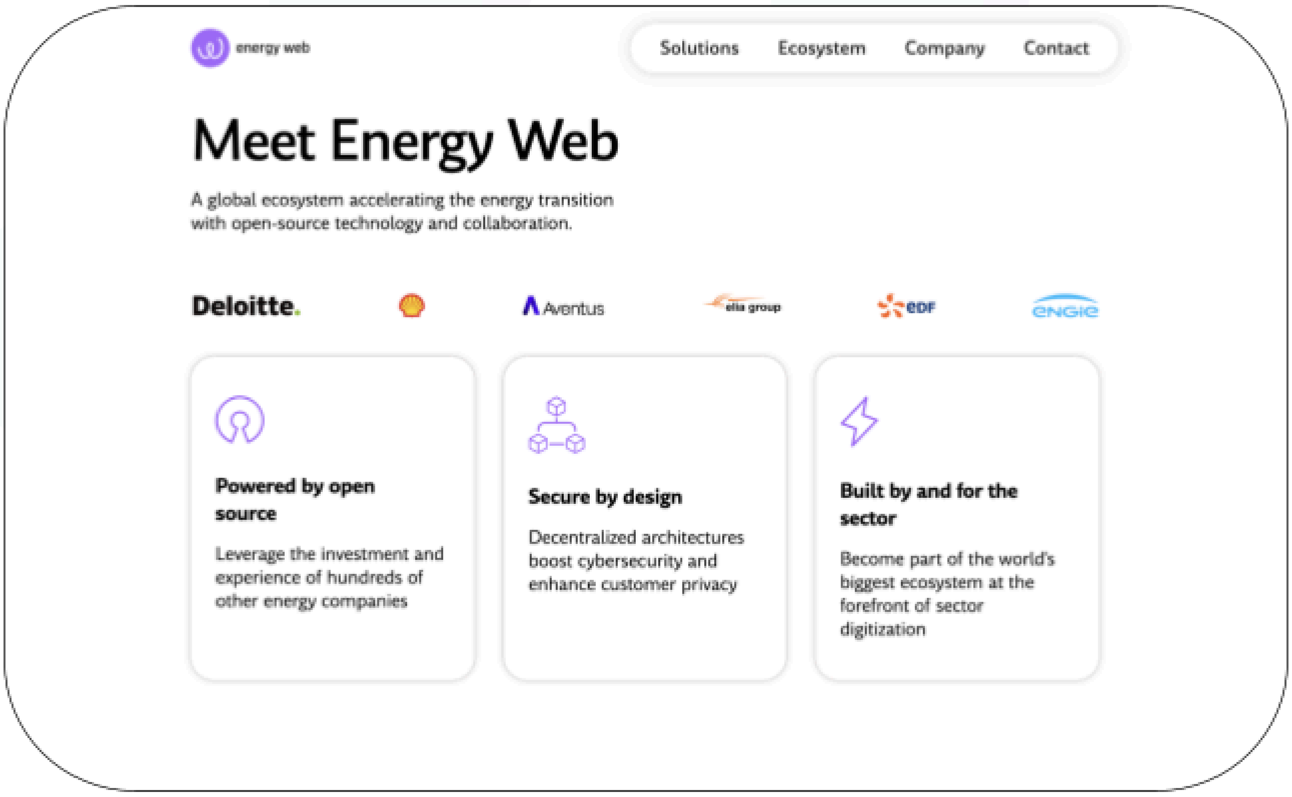
energy web

Solutions Ecosystem Company Contact

Build
Connect
Transform

Creating solutions for

Electricity Aviation Maritime E-mobility Clean Mining



energy web

Solutions Ecosystem Company Contact

Meet Energy Web

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

Deloitte. Aventus elia group EDF ENGIE

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

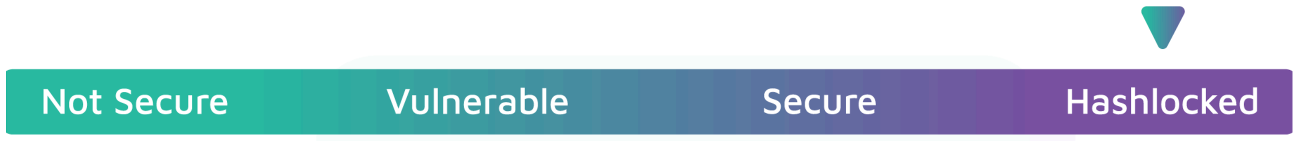
Audit Scope

We at Hashlock audited the Rust code within the Energy Web project, the scope of work included a comprehensive review of the parachain code listed below. We tested the chain 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 Parachain**
Platform	**Polkadot / Rust**
Audit Date	**August, 2025**
Scope	https://github.com/energywebfoundation/energy-web-parachain-node
Audited GitHub Commit Hash (Pull 158)	d4de4a2c600d6620b10252960991ce06c3dd33a9
Audited GitHub Commit Hash (Pull 196)	e9fd6518150db02dcfb830fab2bf91a0c2c286cf
Audited GitHub Commit Hash (Pull 197)	f2bf681c76607b3171a961e4eaf7e652224e4c2d
Fix Review GitHub Commit Hash	b803cc36494afb47fdfe8e6165180de91a98f0e3

Security Rating

After Hashlock's Audit, we found the parachain code to be **"Hashlocked"**. The chain follows complex logic, however, 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 Parachain Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

7 Medium severity vulnerabilities

8 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 Parachain Functions

Claimed Behaviour	Actual Behaviour
parachain-staking pallet Allows nominators to: - Nominate collator candidates with tokens to earn staking rewards - Bond extra tokens or schedule unbonding from collators - Schedule/execute/cancel revocation of nominations after the timelock - Exit all positions at once with scheduled leave Allows collator candidates to: - Join/leave the candidate pool with bonded tokens - Adjust self-stake (bond extra or schedule unbond) - Go offline/online without unbonding - Execute or cancel scheduled requests after the time lock Allows governance/root to: - Set collators per era and era length - Configure minimum stake requirements and delays - Force new era transitions Automatically handles: - Era transitions and collator selection by total stake - Delayed reward distribution and growth period accumulation - Block authorship tracking for rewards - Kicking lowest nominations when limits are reached	**Parachain achieves this functionality.**

Ethereum-bridge pallet Allows users or dapps to: - Submit bridge requests with bounded parameters - Query request and transaction status, and view lifecycle events - Cancel or resubmit failed or pending requests where policy allows Allows validators or OCW to: - Discover Ethereum logs across configured block ranges and partition events for voting - Vote on detected events and corroborate requests - Build, sign, and submit Ethereum transactions - Execute or cancel scheduled actions Allows governance or root to: - Configure bridge contract address, chain ID, network, gas, fee settings, and timeouts - Manage validator thresholds and operational limits - Force processing steps and pause the bridge Automatically handles: - Queueing of new requests and assignment of a sender - Event partitioning, voting tally, and finalization once thresholds are met - Nonce tracking, transaction broadcasting, and receipt polling - Emission of detailed events for auditability and state tracking	**Parachain achieves this functionality.**

Energy Web Parachain configuration & chain specification - Configuration of parachain - para ID, genesis allocations, initial authorities & session keys, enabled pallets. - Configuration of node services - networking, consensus, RPC, telemetry, metrics, database, execution, task spawning. - Configuration of AvnProxy - permitted pallets, calls, signature verification rules, proxy key types. - Configuration of Ethereum helper - RPC endpoints, chain ID, network, signing, broadcast policy, timeouts, retries. - Operations via CLI commands - build and export chain-spec, key management, purge and import blocks, run collator.	**Parachain achieves this functionality.**

Code Quality

This audit scope involves the parachain code of the Energy Web 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. However, not all comments are correctly aligned with code functionality.

Audit Resources

We were given the Energy Web project parachain code in the form of Github access.

As mentioned above, code parts are well commented. The logic is complex, and therefore it is time consuming 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] parachain-staking#select_top_candidates - Root misconfiguration can brick era transitions

Description

The runtime allows `TotalSelected` to be configured above the compiled `MaxCandidates` bound. At era transition, the pallet writes the selected set into `SelectedCandidates`, a bounded vector sized by `MaxCandidates`. If `TotalSelected > MaxCandidates`, converting the chosen set to the bounded type fails and the code uses `expect`, causing a runtime panic.

Vulnerability Details

The `set_total_selected` only checks `new <= Era.length` and not `new <= MaxCandidates`. Later, `<SelectedCandidates<T>>::put(BoundedVec::try_from(collators).expect()` will panic if the chosen set exceeds the bound.

```
pub fn select_top_candidates(now: EraIndex) -> (u32, u32, BalanceOf<T>) {
            let (mut collator_count, mut nomination_count, mut total) =
                (0u32, 0u32, BalanceOf::<T>::zero());
            // choose the top TotalSelected qualified candidates, ordered by stake
            let collators = Self::compute_top_candidates();
            if collators.is_empty() {
                    // SELECTION FAILED TO SELECT >=1 COLLATOR => select collators from
previous era
                let last_era = now.saturating_sub(1u32);
                    let mut total_per_candidate: BTreeMap<T::AccountId, BalanceOf<T>> =
BTreeMap::new();
                // set this era AtStake to last era AtStake
                for (account, snapshot) in <AtStake<T>>::iter_prefix(last_era) {
                    collator_count = collator_count.saturating_add(1u32);
                    nomination_count =
                            nomination_count.saturating_add(snapshot.nominations.len() as
u32);
                    total = total.saturating_add(snapshot.total);
                    total_per_candidate.insert(account.clone(), snapshot.total);
                    <AtStake<T>>::insert(now, account, snapshot);
                }
```

Impact

A misconfiguration (malicious or accidental) halts era transitions with a runtime panic, risking liveness and payouts.

Recommendation

Enforce `new <= MaxCandidates` in `set_total_selected` or change the selection writer to return a controlled error on overflow, not panic through `expect`.

Status

Resolved

[M-02] parachain-staking#nominator_schedule_revoke_all - nominator_schedule_revoke_all reports success even if none of the revokes were scheduled

Description

The function always emits `NominatorExitScheduled` after looping, even if none of the per-collator requests were actually added, for example, all `try_push` operations failed.

Vulnerability Details

The flow iterates the nominator's nominations and tries to enqueue a revoke for each collator, however per-collator queue push errors are ignored. An unconditional "exit scheduled" event is emitted afterwards, even if some pushes failed.

```
<NominatorState<T>>::insert(nominator.clone(), state);

        Self::deposit_event(Event::NominatorExitScheduled {

            era: now,

            nominator,

            scheduled_exit: when,

        });
```

Impact

Users and tooling can act on a misleading event, assuming a full exit was scheduled, while some nominations remain and continue locking funds.

Recommendation

Track whether all revokes were actually enqueued; if any enqueue fails, revert the whole call (or emit a distinct partial-success event and return a non-success error).

Status

Resolved

[M-03] parachain_staking#decrease_bottom_nomination - Bottom-nomination `total` never decreases on `Decrease`

Description

Decreasing a bottom nomination updates the list and ordering, but doesn't adjust the "bottom total" accumulator, causing inflated totals.

Vulnerability Details

When a bottom nomination is decreased, the per-candidate `bottom_nominations.total` is not reduced by the decreased amount.

Top nominations do fix `total`, when bottom nominations do not:

- Top decrease - `top_nominations.total = top_nominations.total.saturating_sub(less);`
- Bottom decrease - updates each bond amount, resorts, resets metadata, but never subtracts `less` from `bottom_nominations.total`.

Later, when a collator exits (`execute_leave_candidates`), the pallet computes the collator's `total_backing` as:

```
state.bond + top_nominations.total + bottom_nominations.total
```

and then subtracts that entire amount from the global `Total`. Because the bottom `total` never went down during decreases, it can be artificially inflated. On exit, `Total` is reduced by too much (potentially to zero via saturation).

An attacker (a nominator or a set of nominators) can repeatedly:

1. increase a bottom nomination (which adds to `bottom_nominations.total`), then
2. schedule & execute multiple decreases (which don't subtract from that `total`).
3. Repeat enough times and the bottom "total" grows far beyond the real bonded amount. When the collator finally exits, `Total` is slashed by the inflated sum due to the code path above. This does not steal funds (locks are removed correctly from accounts), but it corrupts accounting, misreports events, and feeds a wrong "total staked" snapshot used elsewhere.

Impact

Totals used by selection and reward logic diverge from reality, causing minor reward misallocation and confusing UI.

Recommendation

Decrease the bottom total by the same amount before writing the updated nominations back.

Status

Resolved

[M-04] avn_parachain#process_ethereum_events_partition - Missing slashing for invalid ethereum event votes

Description

Validators submitting invalid votes for Ethereum events face no economic penalties, making attacks cheap and reducing the security of the event consensus mechanism.

Vulnerability Details

In `process_ethereum_events_partition`, validators who submit invalid votes are only logged with a comment "`// TODO raise offences`". The `CorroborationOffence` mechanism exists for transaction corroboration but has not been extended to event voting.

Without slashing, malicious validators can repeatedly submit invalid events or vote for wrong partitions without consequences.

```
// Cleanup

        for (partition, votes) in EthereumEvents::<T>::drain() {

            // TODO raise offences

               log::info!("Collators with invalid votes on ethereum events (range: {:?},
partition: {}): {:?}", partition.range(), partition.partition(), votes);

        }
```

Impact

Cheap attacks on event consensus by submitting invalid votes. Potential for vote manipulation if enough validators collude without fear of slashing. Reduced security guarantees for event processing.

Recommendation

Implement an `EventVotingOffence` similar to `CorroborationOffence`. Define clear slashing conditions for invalid event votes. Additionally, add gradual penalties that increase with repeated offenses.

Status

Resolved

[M-05] avn-service#start - Unauthenticated local RPC allows arbitrary signing and sending

Description

The local HTTP service exposes endpoints that allow arbitrary signing with the node's Ethereum private key and sending of arbitrary Ethereum transactions without any authentication, potentially leading to complete compromise of the node's Ethereum wallet.

Vulnerability Details

The HTTP server in `lib.rs` exposes critical endpoints without authentication:

1. `GET /eth/sign/:data_to_sign` signs arbitrary data with the node's Ethereum private key,
2. `POST /eth/send` builds, signs, and broadcasts arbitrary Ethereum transactions.

While the server binds to `127.0.0.1`, this provides insufficient protection. Browser-based attacks using DNS rebinding or malicious JavaScript can bypass same-origin policies. The GET endpoint for signing is particularly vulnerable to CSRF attacks.

Any local malware or compromised process can directly access these endpoints. The `/eth/send` endpoint accepts SCALE-encoded `EthTransaction` objects that specify any destination address and calldata, allowing attackers to transfer funds, approve tokens, or interact with any contract using the node's identity.

```
app.at("/eth/sign/:data_to_sign").get(
        |req: tide::Request<Arc<Config<Block, ClientT>>>| async move {
            log::info!("⛓️  avn-service: sign Request");
            let secp = Secp256k1::new();
            let keystore_path = &req.state().keystore_path;
            let data_to_sign = req.param("data_to_sign")?;
            let hashed_message =
                hash_with_ethereum_prefix(&data_to_sign.to_string()).map_err(|e| {
                    server_error(format!("Error converting data_to_sign into hex string {:?}", e))
                })?;

            log::info!(
                "⛓  avn-service: data to sign: {:?},\n hashed data to sign: {:?}",
                data_to_sign,
                hex::encode(hashed_message)
            );
            let my_eth_address = get_eth_address_bytes_from_keystore(keystore_path)?;
            let my_priv_key = get_priv_key(keystore_path, &my_eth_address)?;

            let secret = SecretKey::from_slice(&my_priv_key)?;
            let message = secp256k1::Message::from_digest_slice(&hashed_message)?;
            let signature: Signature = secp.sign_ecdsa_recoverable(&message, &secret).into();

            Ok(hex::encode(signature.encode()))
        },
    );
```

Impact

Complete compromise of the node's Ethereum wallet, allowing attackers to steal all ETH and tokens, sign arbitrary messages that could be used for off-chain authorization systems, approve malicious contracts to spend tokens, and interact with DeFi protocols using the node's identity.

Recommendation

Implement authentication using HMAC signatures or bearer tokens stored securely. Replace GET endpoints with POST to prevent CSRF via URL embedding. Add Origin and Host header validation to prevent DNS rebinding.

Status

Resolved

[M-06] web3_utils#build_raw_transaction - Missing Chain ID in transaction parameters

Description

Ethereum transactions are signed without explicitly setting the chain ID, allowing replay attacks across different EVM networks and causing transactions to fail or execute on unintended chains.

Vulnerability Details

In `web3_utils.rs::build_raw_transaction`, the `TransactionParameters` struct is created with `Default::default()` for remaining fields, leaving `chain_id` as `None`. Without explicit chain ID, the transaction signing depends on library defaults and RPC configuration, which may not include EIP-155 replay protection.

If the web3 library's behavior changes or the RPC endpoint is misconfigured, transactions could be signed for the wrong network. This is particularly dangerous during network forks or when operating across multiple EVM chains. The `get_chain_id` function exists but is marked as dead code and never used.

```
pub async fn build_raw_transaction(
    web3_data: &mut Web3Data,
    send_request: &EthTransaction,
    sender_eth_address: &Vec<u8>,
) -> anyhow::Result<TransactionParameters> {
    let recipient = send_request.to.as_bytes();

    let nonce = web3_data.get_nonce(sender_eth_address, false).await?;
    let web3 = web3_data.get_web3_instance()?;
    let gas_estimate =
        estimate_gas(web3, sender_eth_address, recipient, &send_request.data).await?;

    Ok(TransactionParameters {
        nonce: Some(nonce.into()),
        to: Some(H160::from_slice(recipient)),
        value: U256::zero(),
        gas: gas_estimate,
        gas_price: None,
        data: web3::types::Bytes(send_request.data.clone()),
        ..Default::default()
    })
}
```

Impact

Signed transactions could be replayed on other EVM networks where the same address has funds, potentially causing unintended transfers or contract interactions.

Recommendation

Fetch the chain ID once during service initialization using the existing `get_chain_id` function. Store it as a service state and explicitly set it in every `TransactionParameters` before signing.

Status

Resolved

[M-07] parachain-staking#nomination_schedule_{revoke/decrease} - Scheduled nomination requests can be silently dropped when the per-collator queue is full

Description

When a collator’s request queue is at capacity, new nomination requests (revoke/decrease) are discarded without failing the extrinsic. Callers see success even though nothing was queued.

Vulnerability Details

Both scheduling flows attempt to append to the per-collator `NominationScheduledRequests`. On capacity error, they do nothing and still return `Ok`, so the caller cannot tell if the action failed. This is visible in the revoke and decrease paths where the “push” failure branch is ignored and the dispatchable still completes successfully.

```
pub(crate) fn nomination_schedule_revoke(
        collator: T::AccountId,
        nominator: T::AccountId,
    ) -> DispatchResultWithPostInfo {
                                                      let       mut       state       =
<NominatorState<T>>::get(&nominator).ok_or(<Error<T>>::NominatorDNE)?;
        let mut scheduled_requests = <NominationScheduledRequests<T>>::get(&collator);

        ensure!(
            !scheduled_requests.iter().any(|req| req.nominator == nominator),
            <Error<T>>::PendingNominationRequestAlreadyExists,
        );
```

Impact

An attacker (or just normal load) can fill a collator’s queue so subsequent nominators believe they’ve scheduled changes when they haven’t, and funds remain locked longer than expected.

Recommendation

Return an explicit error when the queue is full. Additionally, emit a failure event and add per-nominator limits or increase capacity so the queue can’t be trivially saturated.

Status

Resolved

Low

[L-01] parachain-staking#nomination_scheduled_requests - Per-collator request queues are limited by per-nominator constant

Description

The per-collator queue `NominationScheduledRequests` is bounded by `T::MaxNominationsPerNominator`. That constant is about how many nominations one nominator may hold, not how many nominators can line up requests against a single collator. This is a semantic mismatch that makes the queue far smaller than it should be for busy collators:

```
/// Stores outstanding nomination requests per collator.
    #[pallet::storage]
    #[pallet::getter(fn nomination_scheduled_requests)]
    pub(crate) type NominationScheduledRequests<T: Config> = StorageMap<
        _,
        Blake2_128Concat,
        T::AccountId,
                                      BoundedVec<ScheduledRequest<T::AccountId,    BalanceOf<T>>,
T::MaxNominationsPerNominator>,
        ValueQuery,
    >;
```

Storage uses a `BoundedVec<..., T::MaxNominationsPerNominator>` keyed by collator, not by nominator. That means a single collator's queue can be capped by a value intended for a different dimension entirely.

Recommendation

Introduce a dedicated `MaxScheduledRequestsPerCollator` constant and migrate storage to use it.

Status

Acknowledged

[L-02] parachain_staking#rm_top_nomination - Inverted change flag from "remove top nomination"

Description

The helper that removes a nomination from the top list reports whether "total changed," but the returned boolean is inverted.

The function returns `true` when nothing changed and `false` when it did, which is backwards. This risks callers skipping necessary updates.

```
// update candidate info

        self.reset_top_data::<T>(candidate.clone(), &top_nominations);

        self.nomination_count = self.nomination_count.saturating_sub(1u32);

        <TopNominations<T>>::insert(candidate, top_nominations);

        // return whether total counted changed

        Ok(old_total_counted == self.total_counted)
```

Recommendation

Swap the boolean logic to return `true` when the total actually changed; add a unit test that mutates state and asserts the flag.

Status

Unresolved

[L-03] parachain-staking#Genesis - Genesis build double-counts new candidates in candidate_count

Description

The genesis `build` flow increments the candidate counter twice for a successful join, overstating the count.

After inserting a new candidate, the local counter is incremented twice as the loop continues, which overestimates the number of candidates.

```
candidate_count = candidate_count.saturating_add(1u32);
                if let Err(error) = <Pallet<T>>::join_candidates(
                    T::RuntimeOrigin::from(Some(candidate.clone()).into()),
                    balance,
                    candidate_count,
                ) {
                        log::warn!("Join candidates failed in genesis with error {:?}",
error);
                } else {
                    candidate_count = candidate_count.saturating_add(1u32);
                }
```

Recommendation

We recommend incrementing `candidate_count` once per successful join.

Status

Unresolved

[L-04] parachain_staking#signed_schedule_revoke_nomination - Wrong error enum on signature failure

Description

The wrong error enum is used in the signature check path for `signed_schedule_revoke_nomination`.

The signed version of `signed_schedule_revoke_nomination` returns `UnauthorizedSignedRemoveBondTransaction` on signature failure, which refers to a different action. This confuses operators and tooling.

```
ensure!(
                                  verify_signature::<T::Signature, T::AccountId>(&proof,
&signed_payload.as_slice())
                    .is_ok(),
                Error::<T>::UnauthorizedSignedRemoveBondTransaction
            );


            Self::nomination_schedule_revoke(collator, nominator.clone())?;
```

Recommendation

Use a revoke-specific error type or a generic "Unauthorized" error shared by all signed flows.

Status

Unresolved

[L-05] parachain-staking#add_nomination - Underlying nominations storage hard-limits at 300, but capacity checks use configurable limits

Description

The nomination storage uses a hard-coded limit of 300, while capacity checks use runtime-configurable limits. When runtime configuration exceeds 300, the system reports available capacity but silently fails on insertion, corrupting the total vs list accounting invariant.

The base storage type enforces a hard limit:

```
pub type MaxNominations = ConstU32<300>;
```

However, capacity checks use runtime-configurable parameters:

```
x if x.len() as u32 >= T::MaxTopNominationsPerCandidate::get() => CapacityStatus::Full,
```

When inserting nominations, the code updates totals BEFORE attempting insertion, then silently ignores failures:

```
top_nominations.total = top_nominations.total.saturating_add(more);

top_nominations.sort_greatest_to_least();
```

If `MaxTopNominationsPerCandidate > 300`, the system reports capacity available, updates the total, but fails to insert the nomination, creating a divergence between the sum and actual nominations.

Recommendation

Ensure `BoundedVec` type parameters align with runtime configuration limits. Add runtime validation that `MaxTopNominationsPerCandidate` and `MaxBottomNominationsPerCandidate` never exceed 300.

Status

Unresolved

[L-06] parachain-staking#hotfix_remove_nomination_requests_exited_candidates - Wrong error emitted in hotfix function

Description

The hotfix function uses `InsufficientBalance` error when validating the length of the candidates array, which is semantically incorrect and misleading.

The function checks that `candidates.len` is less than 100 but returns `Error::InsufficientBalance` on failure. This error type is meant for balance-related failures, not array length validation. This creates confusion during debugging and makes error handling inconsistent across the codebase.

Recommendation

Introduce a dedicated error variant, such as `TooManyItems` or `InvalidInputLength` for array size validation. Update all similar validation checks to use semantically appropriate error types.

Status

Unresolved

[L-07] avn_parachain#remove_active_request_impl - Unrestricted admin active request removal

Description

The `remove_active_request_impl` function accessible through `AdminSettings::RemoveActiveRequest` allows the admin to remove any active request without any state validation. The function doesn't check whether an Ethereum transaction was already sent (`eth_tx_hash != H256::zero()`), whether corroborations are in progress, or the transaction's expiry status.

Most critically, it ALWAYS notifies the calling pallet with `false` for `Send` requests and `Err` for `LowerProof` requests, forcing a failure state regardless of the actual Ethereum transaction outcome. If a request is removed after `add_eth_tx_hash` is called but before corroborations complete, and the Ethereum transaction succeeds, funds will be permanently locked on Ethereum while Substrate rolls back its state due to the forced failure notification.

The function also ignores the return value of notification callbacks using `let _ =`, providing no indication if the rollback itself failed.

Recommendation

Implement comprehensive state validation before allowing request removal. Check if `eth_tx_hash` is set, verify no pending corroborations exist, and ensure the request has exceeded reasonable timeout periods (like 2x expiry time).

Status

Acknowledged

[L-08] runtime/lib.rs - Zero existential deposit enables state bloat attacks

Description

The existential deposit is configured to zero in non-benchmark builds, allowing creation of unlimited dust accounts without any economic cost, enabling severe state bloat attacks.

Vulnerability Details

The configuration sets `EXISTENTIAL_DEPOSIT: Balance = 0` for normal runtime builds. This means accounts can be created with zero balance, allowing attackers to generate a large amount of empty accounts at no cost beyond transaction fees.

Each account entry consumes storage space and increases the state size that all nodes must maintain. The only protection is transaction fees, which may be insufficient to prevent state growth attacks.

```
/// The existential deposit. Set to 1/10 of the Connected Relay Chain.

#[cfg(not(feature = "runtime-benchmarks"))]

pub const EXISTENTIAL_DEPOSIT: Balance = 0;
```

Impact

Attackers can bloat the blockchain state by creating a large number of dust accounts, degrading network performance for all users. Node operators face increased storage requirements and slower synchronization times

Recommendation

Set a meaningful existential deposit value or calculate based on actual storage costs. Implement account reaping for accounts below the existential deposit. Consider different values for different account types if needed.

Status

Acknowledged

QA

[Q-01] parachain_staking#execute_leave_candidates - Optimistic lock cleanup on invariant failure

Description

In the recovery path, the code unconditionally removes the nominator lock to ensure cleanup, which could mask deeper issues if storage drift occurs.

During `execute_leave_candidates`, if an unexpected inconsistency is encountered (nominator state missing the recorded nomination), the function forcibly clears the nominator lock.

The comment acknowledges this as a "TODO: review" assumption. While not exploitable in normal flows, it is not a good security practice.

Recommendation

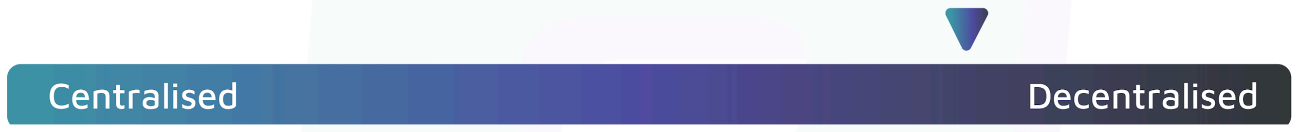
Replace this code part with defensive checks and targeted remediation. Additionally, emit an alarm event and avoid unilateral lock removal when invariants fail.

Status

Acknowledged

Centralisation

The Energy Web Foundation 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 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 parachain 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 the parachain code in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the parachain 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 parachain.

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

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

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