MONETHIC

Acurast Chain

Blockchain Security Audit

Prepared for:
Acurast Association

Date:
23.10.2025

Version:
Final, v1.1

Table of Contents

About Monethic

Monethic is a young and thriving cybersecurity company with extensive experience in various fields, including Smart Contracts, Blockchain protocols (layer 0/1/2), wallets and off-chain components audits, as well as traditional security research, starting from penetration testing services, ending at Red Team campaigns. Our team of cybersecurity experts includes experienced blockchain auditors, penetration testers, and security researchers with a deep understanding of the security risks and challenges in the rapidly evolving IT landscape. We work with a wide range of clients, including fintechs, blockchain startups, decentralized finance (DeFi) platforms, and established enterprises, to provide comprehensive security assessments that help mitigate the risks of cyberattacks, data breaches, and financial losses.

At **Monethic**, we take a collaborative approach to security assessments, working closely with our clients to understand their specific needs and tailor our assessments accordingly. Our goal is to provide actionable recommendations and insights that help our clients make informed decisions about their security posture, while minimizing the risk of security incidents and financial losses.

About Project

Acurast is a decentralized, serverless compute network built on Substrate that turns smartphones into confidential workers by running jobs inside phone TEEs, and it's grown to tens of thousands of active devices. Through its Hyperdrive stack, Acurast provides bidirectional cross-chain messaging and deploys proxy contracts so users on external chains can create deployments and reward processors in native tokens.

The network is integrated with ecosystems like Aleph Zero and Vara, enabling dApps on those chains to offload compute to Acurast's decentralized cloud.

Disclaimer

This report reflects a rigorous security assessment conducted on the specified product, utilizing industry-leading methodologies. While the service was carried out with the utmost care and proficiency, it is essential to recognize that no security verification can guarantee 100% immunity from vulnerabilities or risks.

Security is a dynamic and ever-evolving field. Even with substantial expertise, it is impossible to predict or uncover all future vulnerabilities. Regular and varied security assessments should be performed throughout the code development lifecycle, and engaging different auditors is advisable to obtain a more robust security posture.

This assessment is limited to the defined scope and does not encompass parts of the system or third-party components not explicitly included. It does not provide legal assurance of compliance with regulations or standards, and the client remains responsible for implementing recommendations and continuous security practices.

Scoping Details

The purpose of the assessment was to conduct a Blockchain Security Audit against Acurast Substrate Pallets, shared with the Monethic through the GitHub platform and selected *b04d40b7a9755824701f594d7ae607b15d987f1f* commit hash.

Scope

The scope of the assessment includes the files listed below:

- `p256-crypto`
- `pallets/acurast (excluding p384 libs)`
- `pallets/acurast/common`
- `pallets/candiate-preselection`
- `pallets/compute`
- `pallets/hyperdrive-ibc`
- `pallets/hyperdrive-token`
- `pallets/hyperdrive`
- `pallets/marketplace`
- `pallets/processor-manager`
- `pallets/rewards-treasury`
- `runtime/acurast-mainnet`
- `runtime/common`

GitHub repository:

- https://github.com/Acurast/acurast-substrate/

Timeframe

On 21.08.2025 Monethic was requested for Acurast Substrate Pallets security review. Work began 16.09.2025.

On 03.10.2025, the report from the Blockchain Security assessment was delivered to the Customer.

Between 13.10.2025 and 21.10.2025 the fix review was performed by the Monethic team. On 21.10.2025, the Final Report was shared with the Customer.

Vulnerability Classification

All vulnerabilities described in the report were thoroughly classified in terms of the risk they generate in relation to the security of the contract implementation. Depending on where they occur, their rating can be estimated on the basis of different methodologies.

In most cases, the estimation is done by summarizing the impact of the vulnerability and its likelihood of occurrence. The table below presents a simplified risk determination model for individual calculations.

		Impact		
	Severity	**High**	**Medium**	**Low**
Likelihood	**High**	**Critical**	**High**	**Medium**
	Medium	**High**	**Medium**	**Low**
	Low	**Medium**	**Low**	**Low**

Vulnerabilities that do not have a direct security impact, but may affect overall code quality, as well as open doors for other potential vulnerabilities, are classified as **Informational**.

Vulnerabilities summary

No.	Severity	Name	Status
1	Critical	Anyone can delete and replay non-expired incoming messages	Resolved
2	High	Unbenchmarked weight calculation enabling inexpensive DoS attack	Resolved
3	High	Incomplete RSA signature verification in attestation validation	Resolved
4	High	Flawed report validation enables premature payouts and reputation inflation	Resolved
5	High	Missing sender-contract validation for Substrate inbound messages	Resolved
6	High	Oracle signature threshold can be met with duplicate keys	Resolved
7	Medium	Missing expiry validation in attestation verification	Acknowledged
8	Medium	Missing `check_scheduling_window` implementation	Acknowledged
9	Medium	Lack of assignment check in advertisement	Acknowledged
10	Medium	Weak and overly restrictive IPFS script validation	Resolved
11	Medium	Missing reputation penalties for unreported executions	Acknowledged
12	Medium	The `finalize_job` extrinsic will always complete	Acknowledged
13	Medium	Incorrect bound used for allowed consumers length check	Resolved

14	**Medium**	Incoming message cleanup removes the wrong key	**Resolved**
15	**Medium**	Inconsistent cleanup logic leaks locked capacity and reduces processor availability	**Resolved**
16	**Low**	Integer overflow in storage tracker unlock causes permanent capacity loss	**Resolved**
17	**Low**	Zero fee and zero amount transfers enable persistent storage bloat	**Resolved**
18	**Low**	Orphaned fee holds when re-sending the same sender nonce after TTL expiry	**Resolved**
19	**Low**	Refunds for non-Acurast job owners are collected locally and not forwarded	**Acknowledged**
20	**Low**	Best effort fee transfer leaves residual locked funds	**Resolved**
21	**Low**	Underestimated weight in certificate revocation list updates	**Resolved**
22	**Low**	Incomplete cleanup of execution environments on deregistration	**Acknowledged**
23	**Low**	Cross-proxy nonce collision in IBC message ID	**Resolved**
24	**Low**	Oracle signatures lack domain separation, enabling cross-context replay	**Resolved**
25	**Informational**	Wrong error message emitted in `remove_message`	**Resolved**
26	**Informational**	Centralization concerns	**Resolved**

Technical summary

1. Anyone can delete and replay non-expired incoming messages

Severity: Critical

Status: Resolved

Location

- *pallets/hyperdrive-ibc/src/lib.rs:383*

Description

The *hyperdrive-ibc* pallets's *clean_incoming* extrinsic is callable by anyone and always removes an entry from *IncomingMessages* collection for each provided ID. The *TTL* is only checked to decide whether to also remove data from the lookup map, but it is an optional side effect of the execution.

As a result, anyone can delete and replay a fresh message from the main storage collection.

```
for id in ids.iter() {
    if let Some(message) = <IncomingMessages<T, I>>::get(id) {
        <IncomingMessages<T, I>>::remove(id);
        if message.current_block.saturating_add(T::IncomingTTL::get()) <
current_block {
            <IncomingMessagesLookup<T, I>>::remove(
                &message.message.sender,
                message.message.nonce,
            );
            i += 1;
        }
    }
}
```

Additionally, although token transfers are safeguarded by per-transfer nonces in *pallets/hyperdrive-token*, other Hyperdrive messages routed through *pallets/hyperdrive* lack equivalent nonce or seen-set protection.

This enables malicious actors to replay these messages, causing duplicate state updates, inconsistent application logic, repeated event emissions, and wasted computational resources.

Remediation

We recommend changing the implementation so that the messages are removed from the `IncomingMessages` collection only if they are already expired.

2. Unbenchmarked weight calculation enabling inexpensive DoS attack

Severity: High

Status: Resolved

Location

- `pallets/hyperdrive-ibc/src/lib.rs`
- `runtime/common/src/weight/pallet_acurast_hyperdrive_ibc_weights.rs`

Description

The `clean_incoming` extrinsic processes a list of message IDs while being charged a flat, static weight - `WeightInfo::clean_incoming()`.

In fact, the runtime weight implementation is only a placeholder and statically computes weights as `DbWeight::reads_writes(3, 3)` and does not scale with `ids.len()`.

As a result, the extrinsic underestimates the true execution cost, which involves multiple storage reads and writes per item.

Furthermore, if all items are expired, the extrinsic may return Pays::No, allowing execution with little or no fee. Even though the input is bounded (`ids.len() ≤ 50`), the lack of proper benchmarking and per-item scaling enables attackers to repeatedly invoke this call, inflating block execution time.

Remediation

We recommend performing proper benchmarking of the `clean_incoming` extrinsic.

3. Incomplete RSA signature verification in attestation validation

Severity: High

Status: Resolved

Location

- *acurast/common/src/attestation.rs*

Description

The RSA certificate signature verification in *pallets acurast/common/src/attestation.rs* does not implement full RSASSA-PKCS1 v1.5 checks. When validate encounters the RSA algorithm, it calls *validate_rsa(payload, &cert.signature_value, pbk)*. This function computes *s^e mod n* and compares only the trailing bytes of the result against the SHA-256 digest of the payload.

This approach ignores the RSASSA-PKCS1 v1.5 encoding requirements, which mandate the presence of a *DigestInfo ASN.1* structure and proper *0x00 0x01 FF... 0x00* padding. As implemented, any signature whose modular exponentiation ends with the correct hash value is accepted, even if the preceding padding or *ASN.1* encoding is invalid.

Because *validate_certificate_chain* may process RSA-signed root or intermediate certificates and relies on this routine, malformed RSA signatures could be incorrectly validated, undermining the authenticity of the certificate chain.

Remediation

We recommend replacing the custom implementation with a standards-compliant RSASSA-PKCS1 v1.5 verifier from a vetted cryptographic library.

4. Flawed report validation enables premature payouts and reputation inflation

Severity: High

Status: Resolved

Location

- *pallets/marketplace/src/lib.rs*
- *pallets/marketplace/src/functions.rs*

Description

During the processing of the *report* extrinsic, the *do_report* function increments the assignment's SLA counters and validates timeliness through *update_next_report_index_on_report* and *check_report_is_timely*.

However, this logic only enforces that a report is submitted before the maximum allowed end time, without requiring that the report occur after the execution has actually begun or within the intended reporting window.

Before the first execution starts, the execution index defaults to zero, and the upper-bound condition is satisfied. As a result, repeated calls to report are accepted and increment *sla.met* until the total number of executions is reached.

Each accepted report triggers a payout through the reward manager, which deducts funds from the job's reserved budget and transfers them to the processor's manager.

For assignments with a "single" execution strategy, the job budget can be completely drained before any work is performed. Even under the "competing" strategy, early or misaligned reports can still trigger rewards because only the "not after end plus tolerance" condition is enforced.

This flaw directly impacts job owners, whose reserved budgets can be prematurely depleted without execution, and also distorts system integrity by inflating processor reputation. Each accepted report not only transfers funds but also emits a reported event and updates reputation metrics as though valid results were delivered.

Remediation

We recommend strengthening the validation of report submissions to ensure they fall strictly within the intended execution window.

5. Missing sender-contract validation for Substrate inbound messages

Severity: High

Status: Resolved

Location

- *pallets/hyperdrive/src/lib.rs*
- *pallets/hyperdrive/src/chain/substrate.rs*

Description

On inbound processing, the Hyperdrive pallet accepts messages from *Subject::AlephZero* and *Subject::Vara* without checking that the sender contract equals the configured on-chain counterparts. The match arm only decodes *message.payload* and executes it, i.e., it validates the `chain` but not the contract address (nor selector).

By contrast, the hyperdrive-token pallet does enforce the configured contract for Ethereum and Solana, rejecting mismatches before decoding - it compares *contract_call.contract* against the stored config and errors on mismatch.

In outbound, the Substrate sender does populate the recipient with *Self::aleph_zero_contract()* and *Self::vara_contract()*, proving the configuration exists.

Any payload signed by the oracles and tagged as *AlephZero* or *Vara* will be accepted and executed regardless of which contract emitted it, violating domain separation between proxy contracts.

Remediation

For *Subject::AlephZero(Layer::Contract(call))* and *Subject::Vara(Layer::Contract(call))*, compare *call.contract* with the configured *Self::aleph_zero_contract()* and *Self::vara_contract()* and fail with *InvalidSender* on mismatch, exactly like the Ethereum branch.

6. Oracle signature threshold can be met with duplicate keys

Severity: High

Status: Resolved

Location

- *pallets/hyperdrive-ibc/src/lib.rs*

Description

The *check_signatures* iterates the provided *(Signature, Public)* pairs and increments *valid* for every pair that verifies within the activity window. There is no set of unique public keys, so the same oracle key can be supplied multiple times and be counted multiple times toward *min_signatures*.

```
            let mut valid = 0;
            signatures.into_iter().try_for_each(
                |(signature, public)| -> Result<(), Error<T, I>> {
                    match <OraclePublicKeys<T, I>>::get(public) {
                        None => {
                            not_found.push((signature.0, public.0));
                        },
                        Some(activity_window) => {
                            // valid window is defined inclusive start_block,
exclusive end_block

                            if activity_window.start_block <= current_block
                                && activity_window
                                    .end_block
                                    .map_or(true, |end_block| current_block <
end_block)
                            {
                                if let Some(r) = &relayer {
                                    ensure!(
                                        signature.verify(&(message, r).encode()[..],
&public),
                                        Error::<T, I>::SignatureInvalid
                                    );
                                } else {
                                    ensure!(
                                        signature.verify(&message.encode()[..],
&public),
```

```
                                    Error::<T, I>::SignatureInvalid
                                );
                            };
                            valid += 1;
                        } else {
                            outside_activity_window.push((signature.0,
public.0));
                        }
                    },
                }

                Ok(())
            },
        )?;
```

A single oracle (or minority) can satisfy the threshold by duplicating its own signature entries, subverting the intended multi-party attestation.

Remediation

Deduplicate by public key before incrementing *`valid`*. Maintain a *`BTreeSet<Public>`* of used keys and *`continue`* if *`!inserted`*.

7. Missing expiry validation in attestation verification

Severity: Medium

Status: Acknowledged

Client shared that "*Expiry used to be always validated, but we found out that certain devices (mostly iOS devices) can provide a very short term expiry date (1-3 days) and also most importantly only issue the attestation once per key. For those reasons, we had to change the runtime to check the expiry date of the certificate the attestation is delivered in only at submission. Furthermore, the attestation is used to attest that a key was generated in a secure way and protected by the device's Secure Element / Secure Enclave, and the device guarantees that the key can only be used to generate signatures if the device / app integrity was not compromised.*".

Location

- *`pallets/acurast/src/lib.rs`*

- *pallets/acurast/src/functions.rs*
- *pallets/acurast/src/utils.rs*

Description

The *acurast* pallet enforces attestation validity only at submission time but fails to re-validate expiry during subsequent authorization checks.

Specifically, *submit_attestation* invokes *validate_and_store*, which uses *validate* to call *ensure_not_expired::<T>(&attestation)* before persisting into *StoredAttestation<T>*.

However, later calls to *check_attestation* to retrieve attestations from storage but only verify revocation status and key acceptance. The expiry check (*ensure_not_expired::<T>*) is omitted.

This oversight affects all functions that rely on *check_attestation*, including *ensure_source_verified*, *ensure_source_verified_and_of_type*, and the *EnsureAttested* implementation.

As a result, expired attestations with *Attestation.validity.not_after* already passed may still be accepted as valid, potentially allowing malicious actors to bypass time-based validity constraints.

Remediation

We recommend modifying check_attestation to include an explicit call to *ensure_not_expired::<T>(&attestation)?*. This ensures that on every usage, the current timestamp retrieved via *T::UnixTime::now().as_millis()* is compared against the *not_before* and *not_after* fields of the attestation validity period. Enforcing this check consistently prevents the reuse of expired attestations.

8. Missing *check_scheduling_window* implementation

Severity: Medium

Status: Acknowledged

Location

- *pallets/marketplace/src/match_checker.rs*

Description

In *marketplace* pallet multiple functions rely on *check_scheduling_window* function that is assumed to ensure a specific work can be scheduled. However, it was observed that this function's implementation is commented out and simply returns *Ok(())*.

As a consequence, a matcher can propose matches whose start times fall outside the expected window, violating the consumer's constraint and enabling assignments that should not be allowed.

```
fn check_scheduling_window(
        _scheduling_window: &SchedulingWindow,
        _schedule: &Schedule,
        _now: u64,
        _start_delay: u64,
    ) -> Result<(), Error<T>> {
        //match scheduling_window {
        //    SchedulingWindow::End(end) => {
        //        ensure!(
        //            *end >= schedule
        //                .end_time
        //                .checked_add(start_delay)
        //                .ok_or(Error::<T>::CalculationOverflow)?,
        //            Error::<T>::SchedulingWindowExceededInMatch
        //        );
        //    },
        //    SchedulingWindow::Delta(delta) => {
        //        ensure!(
        //
now.checked_add(*delta).ok_or(Error::<T>::CalculationOverflow)?
        //                >= schedule
        //                    .end_time
        //                    .checked_add(start_delay)
        //                    .ok_or(Error::<T>::CalculationOverflow)?,
        //            Error::<T>::SchedulingWindowExceededInMatch
```

```
        //         );
        //     },
        //}

        Ok(())
    }
```

Remediation

We recommend implementing meaningful scheduling window checks.

9. Lack of assignment check in advertisement

Severity: Medium

Status: Acknowledged

Location

- *pallets/marketplace/src/lib.rs*
- *pallets/marketplace/src/functions.rs*

Description

The *advertise* function explicitly mentions in the comment that currently assigned ads restrict the possible changes to only capacity updates, while also explicitly prohibiting changes to pricing. However, we have observed that no such restrictions are implemented neither in the *advertise* function itself nor in the *do_advertise* function called internally.

In consequence, unexpected pricing changes could occur even when ad is already assigned.

```
       /// Advertise resources by providing a [AdvertisementFor].
        ///
        /// If the source has another active advertisement, the advertisement is
updated given the updates does not
        /// violate any system invariants. For example, if the ad is currently
assigned, changes to pricing are prohibited
        /// and only capacity updates will be tolerated.
        #[pallet::call_index(0)]
        #[pallet::weight(< T as Config >::WeightInfo::advertise())]
```

```
        pub fn advertise(
            origin: OriginFor<T>,
            advertisement: AdvertisementFor<T>,
        ) -> DispatchResultWithPostInfo {
            let who = ensure_signed(origin)?;

            Self::do_advertise(&who, &advertisement)?;

            Self::deposit_event(Event::AdvertisementStored(advertisement, who));
            Ok(().into())
        }
```

Remediation

We recommend implementing a check that would prevent pricing changes when the ad is currently matched.

10. Weak and overly restrictive IPFS script validation

Severity: Medium

Status: Resolved

Location

- *pallets/acurast/common/src/types.rs:30*

Description

The *is_valid_script* function in *pallets/acurast/common/src/types.rs* validates scripts by enforcing a fixed length of 53 bytes and requiring the *ipfs://* prefix.

This check is both insufficient and unnecessarily restrictive. It permits arbitrary data after the prefix, including non-UTF-8 and non-Base58 characters, without verifying whether the content identifier is a properly structured CIFv0. At the same time, it incorrectly rejects valid IPFS links that use CIDv1, such as those starting with bafy, because of the rigid length requirement.

Since the validation logic is used to gate registrations and script edits in *pallets/acurast/src/functions.rs* and *pallets/marketplace/src/lib.rs* , it allows unusable or malformed scripts to pass while blocking valid IPFS references, undermining reliability and interoperability.

```
pub fn is_valid_script(script: &Script) -> bool {
	let script_len: u32 = script.len().try_into().unwrap_or(0);
	script_len == SCRIPT_LENGTH && script.starts_with(SCRIPT_PREFIX)
}
```

Remediation

We recommend strengthening the validation logic to properly check the structure of the CID.

11. Missing reputation penalties for unreported executions

Severity: Medium

Status: Acknowledged

Location

- *pallets/marketplace/src/lib.rs*

Description

The reputation system does not penalize processors who fail to submit reports.

Reputation updates are only triggered through *do_update_reputation*, which is invoked exclusively by the reporting flow. When a job is finalized or an assignment is cleaned up, no additional reputation adjustment is applied for unconsumed SLA slots.

As a result, if a processor accepts an assignment but fails to report some or all executions, the discrepancy between the expected and actual reports has no impact on their reputation.

This allows processors to underperform or skip executions without consequence. At job finalization, reputation reflects only the reports that were submitted, leaving missing reports unpenalized. This enables reputation inflation and gaming: a processor can accept work, provide no results, and still avoid negative reputation effects, undermining the credibility of the scoring system.

Remediation

We recommend extending the finalization and cleanup logic to account for missing reports. The difference between the total SLA executions and the number of reports submitted should result in a proportional penalty applied during job closure.

12. The *finalize_job* extrinsic will always complete

Severity: Medium

Status: Acknowledged

Location

- *pallets/marketplace/src/lib.rs:576*
- *pallets/marketplace/src/functions.rs:253*

Description

The *finalize_job* function calls a *do_cleanup_assignment* function internally and if that call does not error, the *JobFinalized* event will be deposited. However, the *do_cleanup_assignment* function will not error due to business logic checks, as they are implemented via *if let* syntax. This results in this function always returning *Ok(())* regardless if any cleanup logic was executed or not. Furthermore, the *JobFinalized* event contains only the *JobId* that is specified by the caller.

No on-chain damage can be done, however anyone can force the pallet into emitting the *JobFinalized* event with an arbitrarily chosen *JobId*, including one that was not assigned to them. It might lead to confusion and integrity issues, depending on systems and components listening to those events.

Additionally, it must be noted that it's possible the *finalize_job*'s extrinsic implementation is not finished, or is invalid. The *finalize_jobs* extrinsic also present in the same pallet has substantially more complex business logic while also emitting a *JobFinalized* event.

```
pub(crate) fn do_cleanup_assignment(
    processor: &T::AccountId,
    job_id: &JobId<T::AccountId>,
) -> DispatchResult {
    if let Some(assignment) = <StoredMatches<T>>::get(processor, &job_id) {
```

```
            if let Some(job) = <StoredJobRegistration<T>>::get(&job_id.0, job_id.1)
{
                let now = Self::now()?;
                let job_end_time =

job.schedule.actual_end(job.schedule.actual_start(assignment.start_delay))
                        + T::ReportTolerance::get();
                if job_end_time < now {
                    <StoredMatches<T>>::remove(processor, job_id);
                    <AssignedProcessors<T>>::remove(job_id, processor);
                }
            } else {
                <StoredMatches<T>>::remove(processor, job_id);
                <AssignedProcessors<T>>::remove(job_id, processor);
            }
        }
        Ok(())
    }
```

Remediation

Return an *Err* variant in case an *Assignment* or *JobRegistration* are not *Some*. Additionally, make sure that the current logic of *finalize_job* extrinsic is correct.

13. Incorrect bound used for allowed consumers length check

Severity: Medium

Status: Resolved

Location

- *pallets/marketplace/src/functions.rs*

Description

The *do_advertise* function validates the length of *advertisement.allowed_consumers* against a value derived from *T::MaxAllowedSources*.

This is a semantic mismatch, since the *allowed_consumers* field is defined in *pallets/marketplace/src/types.rs* as a bounded vector parameterized by *T::MaxAllowedConsumers*.

On mainnet, `MaxAllowedConsumers` is configured as `100`, while `MaxAllowedSources` is configured as `1000`.

This would create inconsistencies in runtime behavior and potentially permit configurations that violate protocol assumptions.

Remediation

We recommend correcting the validation logic by replacing the reference to `T::MaxAllowedSources` with `T::MaxAllowedConsumers` in `do_advertise`.

14. Incoming message cleanup removes the wrong key

Severity: Medium

Status: Resolved

Location

- `pallets/hyperdrive-ibc/src/lib.rs:355`
- `pallets/hyperdrive-ibc/src/lib.rs:387`

Description

The `receive_message` extrinsic in the `hyperdrive-ibc` pallet saves data into `IncomingMessagesLookup` storage map using the `recipient` and `id` as key tuple. The `id` is constructed as a hash of the message's `sender` and its `nonce`. It was observed that the `clean_incoming` extrinsic is attempting to remove data from this collection if an associated message is expired. However, removal is attempted using the message's `sender` and just its `nonce` which does not match the key used to save it in the first place. As a consequence, the `IncomingMessagesLookup` collection will always be increasing in size.

Remediation

We recommend changing the storage removal implementation so that it uses the storage key matching one used to create a given entry.

15. Inconsistent cleanup logic leaks locked capacity and reduces processor availability

Severity: Medium

Status: Resolved

Location

- *pallets/marketplace/src/lib.rs*

Description

The *cleanup_storage* extrinsic removes assignment indices (*AssignedProcessors* and *StoredMatches*) for a *job_id* when *StoredJobRegistration* is already missing, but it never invokes *StorageTracker::unlock*.

As a result, storage locked during matching (*StorageTracker::lock*) is not released back into the processor's *StoredStorageCapacity*. Since the job metadata is gone at this point, the pallet no longer has enough information to restore capacity, leaving the reduction permanent.

This issue extends beyond the Root-only cleanup path. Both *do_cleanup_assignment* (used in *finalize_job*) and *cleanup_previous_execution_matches* (automatic cleanup in the competing model) also remove *StoredMatches* and *AssignedProcessors* without calling unlock. This means capacity leaks occur not only through manual cleanup but also during normal operation, whenever processors fail to acknowledge or when assignments expire.

While some finalization and deregistration flows correctly call unlock when *StoredJobRegistration* is still available, the inconsistency across cleanup paths creates a systemic loss of recorded capacity. Over time, repeated cleanups can compound the leak across many processors, preventing them from meeting capacity requirements and effectively locking them out of future job assignments.

This creates a long-lived, network-wide reduction in available processor capacity, leading to a gradual denial of service. Unless corrected out of band, affected processors may be permanently unable to accept new jobs despite having sufficient real resources.

Remediation

We recommend ensuring that all cleanup paths consistently restore processor capacity by invoking `StorageTracker::unlock` before removing the assignment state.

16. Integer overflow in storage tracker unlock causes permanent capacity loss

Severity: Low

Status: Resolved

Location

- `pallets/marketplace/src/lib.rs`

Description

The `StorageTracker::unlock` function attempts to increase available storage capacity using `checked_add`.

On overflow, `checked_add` returns `None`, which is written directly back into the `StoredStorageCapacity` state. Despite this failure, the function still returns success, effectively erasing the processor's storage capacity. This allows a malicious or accidental overflow condition to permanently disrupt the affected processor's ability to participate in assignments.

Once overflow occurs, subsequent reports or finalizations leave the victim's capacity entry set to `None`. Any future matching attempts will then fail with `CapacityNotFound`, blocking the processor from being assigned new jobs until they re-advertise.

Remediation

We recommend introducing strict overflow handling in the unlock logic. The function should reject operations that would exceed the maximum representable value, returning an explicit error instead of silently writing `None`.

17. Zero fee and zero amount transfers enable persistent storage bloat

Severity: Low

Status: Resolved

Location

- *pallets/hyperdrive-ibc/src/lib.rs:210*

Description

The *send_test_message* extrinsic is publicly callable, accepts arbitrary fee (including zero) and TTL, and inserts new *OutgoingMessages* without requiring any storage deposit or enforcing a minimum fee.

Attackers can repeatedly send zero-fee, long-TTL messages, cheaply filling *OutgoingMessages* and relayer queues.

Remediation

We recommend enforcing nonzero minimums for both amount and fee in *transfer_native*. Also we recommend requiring either a storage deposit or a nonzero minimum fee for *send_test_message*.

18. Orphaned fee holds when re-sending the same sender nonce after TTL expiry

Severity: Low

Status: Resolved

Location

- *pallets/hyperdrive-ibc/src/lib.rs*

Description

The *do_send_message* function in *pallets/hyperdrive-ibc* allows overwriting an existing *OutgoingMessages* entry once its TTL has expired (*ttl_block < current_block*). However, when this overwrite occurs, the pallet does not release the previously held fee associated with the old message.

Since fee holds are tracked globally under *HoldReason::OutgoingMessageFee* rather than keyed per message, the overwrite erases the storage reference to the old hold. As a result, the original funds remain locked indefinitely and cannot be recovered, leaving orphaned fee holds.

Users or automated processes that resend the same *(sender, nonce)* after TTL expiry will create new holds without releasing the old ones. Over time, this can trap significant funds in unreleasable holds, reducing available balances and causing systemic fund loss.

Remediation

We recommend updating *do_send_message* to explicitly release the existing fee hold when an expired message entry is overwritten. Holds should be keyed per message or include metadata linking them to their originating *OutgoingMessages* entry, ensuring they can always be released when a message is replaced or cleaned up.

19. Refunds for non-Acurast job owners are collected locally and not forwarded

Severity: Low

Status: Acknowledged

Location

- *pallets/marketplace/src/payments.rs:227*

Description

Whenever the job owner is cross-chain, i.e. it is not Acurast, refunds are transferred to a local *hyperdrive* pallet's account with a *TODO* note to later forward it to the proxy chain. However, currently there is no way to send those funds back to the external owner, resulting in stuck refunds.

```
match &job_id.0 {
            MultiOrigin::Acurast(who) => {
                Currency::transfer(
                    &pallet_account,
                    who,
                    remaining.saturated_into(),
```

```
                    Preservation::Preserve,
                )?;
            },
            MultiOrigin::Tezos(_)
            | MultiOrigin::Ethereum(_)
            | MultiOrigin::AlephZero(_)
            | MultiOrigin::Vara(_)
            | MultiOrigin::Ethereum20(_)
            | MultiOrigin::Solana(_) => {
                Currency::transfer(
                    &pallet_account,
                    // TODO refunded amount is collected on
hyperdrive_pallet_account but not yet refunded to proxy chain
                    &hyperdrive_pallet_account,
                    remaining.saturated_into(),
                    Preservation::Preserve,
                )?;
            },
        };
```

Remediation

Implement the outbound refund path through Hyperdrive or explicitly restrict cross-chain registrants until refunds are fully supported end-to-end.

20. Best effort fee transfer leaves residual locked funds

Severity: Low

Status: Resolved

Location

- *pallets/hyperdrive-ibc/src/lib.rs:239*

Description

The *confirm_message_delivery* function in the Hyperdrive IBC pallet uses *transfer_on_hold* with *Precision::BestEffort*, *Restriction::OnHold*, and *Fortitude::Polite*. This configuration allows the transfer to succeed even if less than the full fee is moved.

Afterward, the code unconditionally removes the message from storage. As a result, if only a partial transfer occurs, the leftover held balance remains permanently locked under `OutgoingMessageFee` with no mechanism to release it.

Residual fee holds accumulate when full payout does not occur. These stranded balances cannot be reclaimed or released, leading to permanent loss of funds for affected payers. Over time, multiple such events could degrade system usability by locking balances in inaccessible holds.

Remediation

We recommend providing recovery logic to detect and release orphaned holds left behind by partial transfers.

21. Underestimated weight in certificate revocation list updates

Severity: Low

Status: Resolved

Location

- `pallets/acurast/src/lib.rs`

Description

The root-only extrinsic `update_certificate_revocation_list` in `pallets/acurast/src/lib.rs` applies a batch of updates but charges weight as if only a single storage write were performed. The function accepts updates: `BoundedVec<CertificateRevocationListUpdate, T::MaxCertificateRevocationListUpdates>` and iterates over each entry, either inserting or removing values from `Acurast::StoredRevokedCertificate<T>` (keyed by `SerialNumber`).

Although multiple updates may be processed in a single call, the weight function defined in `runtime/common/src/weight/pallet_acurast.rs` (`fn update_certificate_revocation_list() -> Weight`) assigns a constant cost and accounts for only one database write via `T::DbWeight::get().writes(1)`. Since the runtime configuration sets type `MaxCertificateRevocationListUpdates = frame_support::traits::ConstU32<10>`, a call may perform up to 10 storage modifications while incurring weight charges for only one.

This discrepancy results in systematic underestimation of runtime costs, with up to ~9 additional writes and loop overhead unaccounted for. While the extrinsic is restricted to Root origin, the miscalculated weight introduces inaccuracies in resource usage accounting and may undermine system performance modeling.

Remediation

We recommend modifying the weight function signature to account for the number of updates processed. Additionally, given that the `update_certificate_revocation_list` extrinsic is critical for the system's overall integrity, registering it with `DispatchClass::Mandatory` should be considered.

22. Incomplete cleanup of execution environments on deregistration

Severity: Low

Status: Acknowledged

Location

- `pallets/acurast/src/lib.rs`
- `pallets/acurast/src/functions.rs`

Description

The deregistration process fails to fully remove per-job execution environment entries, leaving orphaned storage records under `Acurast::ExecutionEnvironment`. Environments are stored as a `StorageDoubleMap` keyed by `(JobId<T::AccountId>, T::AccountId)`. These entries are populated via `set_environments`, which accepts `BoundedVec<(T::AccountId, EnvironmentFor<T>), T::MaxSlots>`.

Although each call is bounded by `T::MaxSlots`, repeated invocations allow more than `T::MaxSlots` unique sources to accumulate for a single job. During deregistration, `deregister_for` invokes `clear_environment_for`, which calls `<ExecutionEnvironment<T>>::clear_prefix(job_id, T::MaxSlots::get(), None)`. This clears at most `T::MaxSlots` entries and ignores any remaining cursor, leaving additional entries under the same `job_id` permanently stored.

As a result, orphaned `(JobId, source) -> EnvironmentFor<T>` mappings persist after deregistration, causing unnecessary storage growth and the risk of stale material lingering in the system.

Remediation

We recommend updating the cleanup logic to ensure the complete removal of all environment entries for a given job.

23. Cross-proxy nonce collision in IBC message ID

Severity: Low

Status: Resolved

Location

- `pallets/hyperdrive-ibc/src/lib.rs`
- `pallets/hyperdrive-token/src/lib.rs`

Description

Outgoing IBC messages are deduped by `id = hash((sender, nonce))`. The hyperdrive-token pallet constructs `nonce` as `hash_of(&transfer_nonce)` where `transfer_nonce` is a per-proxy counter and the sender is the constant pallet account.

In IBC, dedup and lookup is indexed by `(sender, nonce)` and the duplicate check consults `OutgoingMessages(id)` for TTL. Two transfers to different proxies that currently share the same numeric `transfer_nonce` will produce the same `nonce` hash, and with the same `sender` (pallet account), they collide on `id`. The second send is rejected with `MessageWithSameNoncePending` until the first's TTL expires.

A potential attacker issuing concurrent transfers to two proxies at the same local nonce can block one of them until TTL elapses, creating a liveness issue on multi-proxy usage.

Severity of an issue was reduced as for now only one proxy is actively supported - it will become problematic once multiple proxies are enabled.

Remediation

Domain-separate the *nonce* by including proxy (and optionally recipient) into the hashed nonce preimage, or move the recipient and proxy into the *message_id* hashing them directly.

24. Oracle signatures lack domain separation, enabling cross-context replay

Severity: Low

Status: Resolved

Location

- *pallets/hyperdrive-ibc/src/lib.rs:404*

Description

The oracle signature verification logic in *check_signatures* validates signatures over raw SCALE-encoded data (*MessageFor* and optionally *(message, relayer)*) without adding a domain separator or chain identifier. This means the signed payloads are not bound to the pallet, protocol, or network where they are used.

Without domain separation, oracle signatures generated in one context (for another pallet, chain, or network that uses an identical struct and SCALE encoding) can be reused here. Since the verifier only checks the raw bytes against oracle keys, the same signature can be replayed across contexts.

An attacker who can obtain valid oracle signatures elsewhere could replay them to satisfy verification in this pallet. This would allow unauthorized message acceptance or illegitimate fee claims, provided the attacker can construct identical SCALE-encoded bytes.

Remediation

We recommend introducing explicit domain separation in signature verification by prefixing the encoded payload with a protocol identifier, chain ID, or pallet-specific domain tag. This ensures oracle signatures are only valid in their intended context and cannot be replayed elsewhere.

25. Wrong error message emitted in *remove_message*

Severity: Informational

Status: Resolved

Location

- *pallets/hyperdrive-ibc/src/lib.rs:301*

Description

The *remove_message* extrinsic in *hyperdrive-ibc* pallet returns an error of type *CouldNotHoldFee* when *Currency::release* call fails. This is incorrect in this context, as there is a *CouldNotReleaseHoldFee* error implemented for this purpose.

```
T::Currency::release(
    &HoldReason::OutgoingMessageFee.into(),
    &message.payer,
    message.fee,
    Precision::BestEffort,
)
.map_err(|_| Error::<T, I>::CouldNotHoldFee)?;
```

Remediation

We recommend changing the return error type to the appropriate one.

26. Centralization concerns

Severity: Informational

Status: Resolved

Location

- Observed in many places throughout the codebase

Description

It was observed that *RootOrigin* (enforced via *ensure_root* function call) is used in many places throughout the codebase for extrinsics implementing administrative operations. Using *RootOrigin* decreases the decentralization aspect of the whole protocol and introduces a single point of failure into the system.

```
Code Blocks -> Language -> Agate with background
```

Remediation

We recommend introducing *Committees* that would be responsible for performing administrative operations.

END OF THE REPORT