MONETHIC

Acurast Chain - compute pallet

Blockchain Security Audit

Prepared for:
Acurast Association

Date:
11.03.2026

Version:
Final, 1.0

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 Compute Pallet, shared with Monethic through the GitHub platform and selected *cedb450f14d45a68c83070b007868552361da879* commit hash.

Scope

The scope of the assessment includes the files listed below

- `pallets/compute`

GitHub repository:

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

Timeframe

On 02.11.2025 Monethic was requested for Acurast Substrate Compute Pallets security review. Work began 10.11.2025.

On 15.11.2025, the report from the Blockchain Security assessment was delivered to the Customer.

On 15.12.2025, the retest of issues found was performed.

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	Broken locking mechanism makes stake effectively unslasheable	Resolved
2	Critical	Minting a commitment NFT to the manager enables impersonation and committer DoS	Resolved
3	Critical	Metric commitment leads to economic damage to the protocol	Acknowledged
4	High	Epoch reward misattribution via mutable manager commitment mapping at distribution time	Acknowledged
5	High	Unbounded delegation scan in *force_end_commitment* enables operator-origin DoS	Resolved
6	High	Redelegation blocking period check is ineffective	Resolved
7	Medium	Stale collator reward persists across zero-inflation epochs, enabling overpayment	Resolved
8	Low	Unbounded iteration over MetricPools in *on_initialize* facilitates chain-level DoS	Resolved
9	Low	Unbounded pool reward ratios allow per-epoch budget overdistribution	Resolved
10	Low	Redundant stake subtraction in *force_end_commitment_for* corrupts global totals	Resolved

Technical summary

1. Broken locking mechanism makes stake effectively unslasheable

Severity: Critical

Status: Resolved

Location

- pallets/compute/src/staking.src:1920

Description

The issue arises because the staking and delegation flows assumes that whatever is "locked" is economically slashable, while in reality only free balance is ever put at risk. When a user establishes or increases stake via the *stake_for* and *delegate_for* functions, both paths end up calling *lock_funds* to enforce the balance check and install the lock.

Inside *lock_funds*, the pallet computes *new_lock_total* based on *DelegatorTotal* plus the requested additional amount and then explicitly compares this against *Currency::total_balance(who)*, under the assumption that also reserved balance can be locked.

If total_balance is high enough, *lock_funds* calls *LockableCurrency::set_lock* with *new_lock_total*, but Substrate locks only restrict the account's free balance - reserved balance is unaffected and remains fully protected for the pallet that reserved it.

Later, during exit and slashing, the pallet unlocks positions through *unlock_funds* and ultimately *unlock_and_slash*. In *unlock_and_slash*, the pallet first unlocks the full nominal stake.amount and then computes maximum_slashable using only *Currency::free_balance(who)*, effectively taking *min(stake.accrued_slash, stake.amount, free_balance)* and burning at most that amount via *Balanced::withdraw* with *Preservation::Expendable*.

Reserved funds are never touched in this path. This creates a mismatch as *stake_for* and *delegate_for* accept "stake" based on free+reserved via *total_balance* and record full stake weight for reward and slashing accounting, but *unlock_and_slash* can only ever slash up to the much smaller free component.

An attacker can therefore first move most of their tokens into the reserved state through another pallet, leaving only a small free remainder, and then call *stake_for* or *delegate_for* so the *lock_funds* check passes because *total_balance* covers the requested amount, the full amount is recorded as stake and contributes to delegation and commitment weight and reward distribution, yet any later slash is capped by the tiny free balance that remains.

The result is a "phantom stake" position where the system believes it has a large slashable collateral, because of the recorded stake and locks, but economically almost none of that stake can actually be burned, enabling near risk-free extraction of inflationary rewards and breaking the core assumption that stake used for security must be fully slashable.

Remediation

In *lock_funds*, validate against free balance, not *total_balance*. If the intent is to require slasheable backing, ensure *new_lock_total* ≤ *free_balance*. Then set the lock.

2. Minting a commitment NFT to the manager enables impersonation and committer DoS

Severity: Critical

Status: Resolved

Location

- pallets/compute/src/lib.rs:733

Description

The *accept_backing_offer* extrinsic incorrectly mints the commitment NFT to the manager (transaction origin) instead of the intended committer, violating ownership invariants and disrupting the workflow. Specifically, the call to *do_get_or_create_commitment_id* uses the manager's address, causing *CommitmentIdProvider::create_commitment_id* to bind the newly created commitment to the manager. This results in the *CommitmentCreated* event being emitted with the manager as the commitment owner.

As a consequence, all mappings (*Backings* and *BackingLookup*) are tied to a commitment owned by the manager. The committer receives no associated

commitment_id, which prevents access to all committer-only flows that rely on *commitment_id_for*. Operations such as *commit_compute*, *stake_more*, *withdraw_commitment*, and delegation targeting are thus rendered inaccessible, resulting in a denial-of-service.

Furthermore, since the manager owns the NFT, they can invoke committer-restricted actions and accrue rewards or incur slashes attributed to the committer's stake, resulting in impersonation and misattribution. A malicious manager can repeatedly exploit this by accepting multiple offers and monopolizing commitment IDs, effectively blocking legitimate committers from progressing.

```
pub fn accept_backing_offer(
	origin: OriginFor<T>,
	committer: T::AccountId,
) -> DispatchResultWithPostInfo {
	let who = ensure_signed(origin)?;
	let manager_id = T::ManagerIdProvider::manager_id_for(&who)?;

	let offer_manager_id =
		BackingOffers::<T, I>::take(committer).ok_or(Error::<T,
I>::NoBackingOfferFound)?;
	ensure!(manager_id == offer_manager_id, Error::<T, I>::NoBackingOfferFound);

	// technically this call allows to reuse a commitment_id NFT here, but the
mechanism to create commitment IDs in this pallet does not yet allow committers to
swap which managers they back (they can do offer-accept-flow at most once)
	let (commitment_id, created) = Self::do_get_or_create_commitment_id(&who)?;
	if created {
		// we always emit this if extrinsic call succeeds, but it's likely to
change in the future so we already emit this separate event for the first time a
commitment_id-NFT is created
		Self::deposit_event(Event::<T, I>::CommitmentCreated(who,
commitment_id));
	}
```

Remediation

We recommend minting the NFT to the committer instead of the manager.

3. Metric commitment leads to economic damage to the protocol

Severity: Critical

Status: Acknowledged

"This is a specific contract between the runtime and the processor. We specified that the processor can provide normal integer values by specifying only one value in the tuple (and setting the second value to 0). If the processor is malicious and able to alter the contract by providing any value in the tuple, then it really does not matter that 0 is an accepted value for the second number since it can just provide arbitrary values."

Location

- `pallets/compute/src/lib.rs:1446-1462`

Description

The *commit_new_metrics* function permits the construction of *FixedU128* values by coercing a zero denominator to 1, instead of rejecting the input. This allows a malicious processor to submit metrics with artificially low or zero denominators, generating arbitrarily large metric values. These inflated metrics are persisted into both *total_with_bonus* and *MetricsEpochSum*, directly influencing the proportional allocation of compute-based rewards during *do_claim*. As a result, attackers can disproportionately dominate a pool's reward share, siphoning funds at the expense of honest participants.

Additionally, the system uses the previous epoch's finalized metrics as both the upper bound for subsequent metric commitments (*validate_max_metric_store_commitments*) and as the baseline for slashing decisions (*do_slash*). By persistently inflating their "actual" metrics each epoch, the attacker increases their permissible commit bounds and avoids triggering underperformance-based slashing. This ensures uninterrupted reward extraction and protects previously awarded payouts from retrospective clawback.

The broader impact includes sustained misallocation of compute reward budgets, compromised fairness in pool participation, skewed metric and pool totals that undermine incentive structures, degradation of economic security due to participant attrition, and reputational harm stemming from visible metric fraud.

```rust
fn commit_new_metrics(
	processor: &T::AccountId,
	manager_id: T::ManagerId,
	metrics: &[MetricInput],
	active: bool,
	cycle: CycleFor<T>,
) -> Option<Vec<(PoolId, (Metric, Metric))>> {
	let epoch = cycle.epoch;

	let mut prev_metrics_sum: Vec<(PoolId, (Metric, Metric))> = vec![];
	for (pool_id, numerator, denominator) in metrics {
		let Some(metric) = FixedU128::checked_from_rational(
			*numerator,
			if denominator.is_zero() { One::one() } else { *denominator },
		) else {
			continue;
		};
```

Remediation

We recommend changing the implementation so that zero-value denominators are rejected with an error.

4. Epoch reward misattribution via mutable manager commitment mapping at distribution time

Severity: High

Status: Acknowledged

"For now, we will not allow for the mapping to be changed, this functionality will be available in the future."

Location

- pallets/compute/src/lib.rs:1406-1423

Description

The pallet uses the current manager-to-commitment mapping at distribution time instead of an epoch-snapshotted mapping.

Because *accept_backing_offer* updates *BackingLookup* immediately and independently of epoch boundaries, the mapping used to attribute the last epoch's rewards can be changed after the epoch has ended.

An attacker can accept a new backing offer right after an epoch ends but before any processor runs *do_commit* for that epoch. The subsequent distribution reads the updated *BackingLookup* and credits the prior epoch's rewards to the newly linked commitment rather than the one that actually backed that epoch.

This enables redirection of rewards to a colluding commitment, causing misattribution, loss for the rightful recipient, and systemic accounting inconsistencies.

Remediation

We recommend introducing an epoch-snapshotted version of *BackingLookup* that records the active manager-to-commitment relationship at epoch boundaries. Reward attribution should rely exclusively on this snapshot to ensure consistency and prevent retroactive manipulation. Additionally, validation checks should prevent *accept_backing_offer* from altering backing relationships retroactively within the same epoch window.

5. Unbounded delegation scan in *force_end_commitment* enables operator-origin DoS

Severity: High

Status: Resolved

Location

- pallets/compute/src/lib.rs

Description

The force_end_commitment extrinsic in the compute pallet delegates to *force_end_commitment_for* using a statically defined weight (*T::WeightInfo::force_end_commitment*), but internally performs an unbounded, full-table iteration over the Delegations storage. Specifically, it executes *Delegations::iter().collect()* and filters entries by the target *commitment_id*.

Since Delegations is structured as a *StorageDoubleMap* with keys (delegator, *commitment_id*), there is no prefix iteration support for filtering by *commitment_id* alone. This results in:

- O(n) iteration over all global delegations,
- O(n) memory allocation to collect the full dataset,
- Multiple storage writes per match (*DelegatorTotal::mutate*, *unlock_funds*, and *Delegations::remove*).

This creates a critical discrepancy between the declared weight and actual computational cost. As delegation volume grows, a single call, despite being limited to OperatorOrigin, can exceed the block's weight or execution time constraints, producing a denial of service (DoS) vector. The risk is further elevated by the in-memory *collect* execution and the density of follow-up writes per matching delegation.

Remediation

We recommend restructuring the storage schema or introducing a reverse index keyed by commitment_id to enable efficient, bounded prefix iteration. Additionally, rework *force_end_commitment_for* to operate in bounded batches with dynamic weight accounting, or expose a new batched extrinsic to safely process large delegation sets over multiple transactions.

6. Redelegation blocking period check is ineffective

Severity: High

Status: Resolved

Location

- pallets/compute/src/staking.rs:1332

Description

In *staking.rs:1332*, the intended rule is no redelegation for *RedelegationBlockingPeriod * Epoch* after the last update. However, the implementation compares the absolute block number *old_delegation.stake.created* to that threshold, not the elapsed time *(current_block - created)*.

Since block numbers quickly exceed the threshold after genesis, the check generally passes, enabling immediate redelegations when not intended.

```
// Only check RedelegationBlockingPeriod is respected if current committer is
not in cooldown, otherwise allow immediate redelegation always
        if old_commitment_stake.cooldown_started.is_none() {
            // check if enough epochs have passed since last update (which lead
to `created` field being reset)
            let blocks_since_created = old_delegation.stake.created;
            ensure!(
                blocks_since_created
                    >=
T::RedelegationBlockingPeriod::get().saturating_mul(T::Epoch::get()),
                Error::<T, I>::RedelegateBlocked
            );
        }
```

Remediation

We recommend replacing the check with the real delta validation:

```
let now = <frame_system::Pallet<T>>::block_number();
ensure!(now.saturating_sub(old_delegation.stake.created)
        >= T::RedelegationBlockingPeriod::get() * T::Epoch::get(), Error::<T,
I>::RedelegateBlocked);
```

7. Stale collator reward persists across zero-inflation epochs, enabling overpayment

Severity: Medium

Status: Resolved

Location

- pallets/compute

Description

Collator rewards are distributed per block using a value stored in *CollatorRewards*, which is intended to be updated at each epoch boundary by *on_initialize* via the *inflate* function.

However, if inflate() returns None (e.g., when *T::InflationPerEpoch == 0*), no update is written to *CollatorRewards*. Consequently, the system continues to use the previous epoch's non-zero reward value, including on the epoch boundary block itself, resulting in unintended reward issuance.

Similarly, if *inflate* yields a non-zero total inflation amount but the computed collator share rounds to zero (e.g., due to *T::InflationCollatorsRatio == 0* or negligible), the call to *store_collators_reward* is skipped due to zero-value write elision. This causes the boundary block to apply a local zero payout, while subsequent blocks fall back to reading and distributing the outdated non-zero reward.

In both scenarios, the reward from a previous epoch "sticks" across an epoch intended to emit no rewards, leading to unauthorized overpayments to collators. This undermines monetary policy, drains the reward distribution balance, and skews inflation allocation.

Remediation

We recommend modifying *store_collators_reward* to persist zero values explicitly to ensure the correct reward state across epochs.

8. Unbounded iteration over *MetricPools* in *on_initialize* facilitates chain-level DoS

Severity: Low

Status: Resolved

Location

- pallets/compute/src/lib.rs:1200-1217

Description

The per-epoch routine iterates over all entries in *MetricPools* and performs multiple database operations per pool, but there is no runtime cap or weight scaling tied to the number of pools processed during *on_initialize*. A malicious or compromised privileged origin can create a very large number of pools. Each epoch, *on_initialize* becomes *O(N)* in the number of pools and can exceed block weight or PoV limits, degrading performance or halting block production entirely. This threatens chain liveness and availability at the consensus level.

```
fn store_staked_compute_reward(
	amount: BalanceFor<T, I>,
	current_epoch: EpochOf<T>,
	last_epoch: EpochOf<T>,
	target_token_supply: u128,
) -> (Weight, BalanceFor<T, I>) {
	// Store stake-based rewards split by pool (to avoid redoing this in every
stake-based-reward-claiming heartbeat)
	let mut unused_amount: BalanceFor<T, I> = Zero::zero();
	let mut weight = Weight::default();

	for (pool_id, pool) in MetricPools::<T, I>::iter() {
		// we have to use the pool's total compute from the last_epoch since
only this one is a completed rolling sum
		let target_weight_per_compute = U256::from(target_token_supply)
			.saturating_mul(U256::from(PER_TOKEN_DECIMALS))
			.saturating_mul(U256::from(
T::TargetCooldownPeriod::get().saturated_into::<u128>(),
				))
.checked_div(U256::from(T::MaxCooldownPeriod::get().saturated_into::<u128>()))
			.unwrap_or(Zero::zero())
			.saturating_mul(U256::from(FIXEDU128_DECIMALS))
.checked_div(U256::from(pool.total.get(last_epoch).into_inner()))
			.unwrap_or(Zero::zero());
```

Remediation

We recommend capping the maximum number of pools to a value that would not degrade chain performance. Alternatively, a batching mechanism can be considered if limiting the number of pools is not desirable.

9. Unbounded pool reward ratios allow per-epoch budget overdistribution

Severity: Low

Status: Resolved

Location

- `pallets/compute/src/lib.rs:1276-1293`

Description

The pallet fails to enforce that the aggregate of per-pool reward ratios remains ≤1, despite *`ComputeBasedRewards`* representing a single global reward budget per epoch. The *`do_claim`* function aggregates individual pool shares and applies clamping only at the level of each manager's claim, without validating global budget adherence across all pools.

This allows a privileged actor (e.g., a pool administrator) to misconfigure the system by defining multiple reward pools with ratios summing to more than 100%. If compute work is driven into these inflated pools and managers invoke *`do_claim`*, the cumulative payout can exceed the intended per-epoch emission. This results in over-distribution of rewards, depleting the pallet's funds and causing future reward distributions and dependent logic to fail.

The outcome includes budget insolvency, denial of service for legitimate claimants, and systemic inconsistency in reward accounting.

Remediation

We recommend introducing a validation mechanism that enforces a strict upper bound of 1.0 on the sum of all pool reward ratios during configuration updates.

10. Redundant stake subtraction in *force_end_commitment_for* corrupts global totals

Severity: Low

Status: Resolved

Location

- pallets/compute/src/lib.rs:1276-1293

Description

In the compute pallet, the *force_end_commitment_for* function introduces a critical accounting error by decrementing TotalStake twice for the same amount. After calculating *total_amount_to_remove* - which includes the committer's own stake and all delegations - the function first applies a saturating_sub directly on *TotalStake*. It then invokes *update_total_stake(StakeChange::Sub(total_amount_to_remove))*, which attempts a second subtraction via *checked_sub*, silently discarding any error.

Depending on the initial state:

- (a) If TotalStake = s ≥ 2r, the result becomes s - 2r, leading to an understatement by r.
- (b) If r ≤ s < 2r, the second subtraction underflows, fails, and TotalStake remains at s - r.
- (c) If s < r, the initial subtraction clamps to zero, and the second operation fails without effect.

These scenarios produce inaccurate *TotalStake* values, either understating or zeroing out the global metric. This corruption undermines the integrity of all downstream processes dependent on stake data, including reward calculations, slashing conditions, governance weight derivation, and off-chain analytics or policy enforcement.

Remediation

We recommend removing the manual saturating_sub and relying solely on the controlled update path via *update_total_stake(StakeChange::Sub(...))*. Ensure that any underflow is surfaced and handled explicitly. This change preserves consistency and prevents double-decrement errors that compromise stake accounting.

END OF THE REPORT