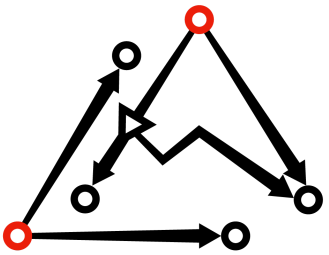

Least Authority

PRIVACY MATTERS

P-DART

Security Audit Report

Polymesh

Final Audit Report: 2 September 2026

Table of Contents

Overview

Background

Polymesh has requested Least Authority perform a security audit of P-DART, a Rust-based confidential assets protocol built for the Polymesh ecosystem.

Project Dates

- **July 21, 2026 - August 21, 2026:** Initial Code Review *(Completed)*
- **August 25, 2026:** Delivery of Initial Audit Report *(Completed)*
- **August 31, 2026:** Verification Review *(Completed)*
- **September 2, 2026:** Delivery of Final Audit Report *(Completed)*

Review Team

- George Gkitsas, Security / Cryptography Researcher and Engineer
- Eduardo Morais, Cryptography Researcher and Engineer
- Miguel Quaresma, Security Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of P-DART (Distributed Anonymous Regulated Transactions) followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories were considered in scope for the review:

- `dart-bp` crate:
  - https://github.com/PolymeshAssociation/polymesh-dart/tree/main/dart-bp
  - All source files under `dart-bp/src/` except unit tests, benchmarks, and the utility files.
- `wrapper` crate:
  - https://github.com/PolymeshAssociation/polymesh-dart/tree/main/src
- Three forked dependencies reviewed only for changes made by Polymesh, not as full audits of the upstream code:
  - https://github.com/PolymeshAssociation/curve-trees
    - `bulletproofs/`
    - `relations/`
    - `ark-ec-divisors/`
    - `ark-dlog-gadget/`
  - https://github.com/PolymeshAssociation/crypto
    - `schnorr_pok/`
    - `utils/`
  - https://github.com/PolymeshAssociation/arkworks-algebra
    - diff vs upstream

Specifically, we examined the following Git revisions for our initial review:

- `Polymesh-dart`: a99ff79470772bdd10c8bd5078ac44e1a347fe5e
- `Crypto`: e7845619a6b1ce85ca9be8722e9d2b8c9ba9edc9
- `Arkworks-algebra`: be949ad980da003b1a43abe81c6d54dbaef8b26a
- `Curve-trees`: ee4e7ea4b6979981c077a3a8117813fef871f96f

For the verification, we examined the following Git revisions:

- `Polymesh-dart`: 7d4b32fac8a553f1840b0b53bc3fb45b23678a44
- `Crypto`: 7e3922f860ed6b6139d4520429cdc60437c1abac
- `Arkworks-algebra`: be949ad980da003b1a43abe81c6d54dbaef8b26a
- `Curve-trees`: 4db58cf137c1ea19daabd05e72dd6d6e12cda7a2

For the review, these repositories were cloned for use during the audit and for reference in this report:

- `PolymeshAssociation-polymesh-dart`: https://github.com/LeastAuthority/PolymeshAssociation-polymesh-dart
- `PolymeshAssociation-crypto`: https://github.com/LeastAuthority/PolymeshAssociation-crypto
- `PolymeshAssociation-arkworks-algebra`: https://github.com/LeastAuthority/PolymeshAssociation-arkworks-algebra
- `PolymeshAssociation-curve-trees`: https://github.com/LeastAuthority/PolymeshAssociation-curve-trees

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Research paper: https://assets.polymesh.network/P-DART-v1.pdf
- Website: https://polymesh.network
- AUDIT_SCOPE.md *(shared with Least Authority via email on 16 June 2026)*.

In addition, this audit report references the following documents:

- F. Bao, R. H. Deng, and H. Zhu, "Variations of Diffie-Hellman problem." *Information and Communications Security: 5th International Conference, ICICS*, 2003, [BDZ03]
- M. Campanelli, M. Hall-Andersen, and S. H. Kamp, "Curve Forests: Transparent Zero-Knowledge Set Membership with Batching and Strong Security." *IACR Cryptology ePrint Archive*, 2024, [CHK24]
- J. H. Cheon, "Security Analysis of the Strong Diffie-Hellman Problem." *IACR Cryptology ePrint Archive*, 2006, [Cheon06]
- A. Chiesa, D. Ojha, and N. Spooner, "Fractal: Post-Quantum and Transparent Recursive Proofs from Holography." *IACR Cryptology ePrint Archive*, 2019, [COS19]
- I. Damgård, "On Σ-protocols." *Aarhus University, Department of Computer Science*, 2010, [Damgård10]
- R. Gennaro, D. Leigh, R. Sundaram, and W. Yerazunis, "Batching Schnorr Identification Scheme with Applications to Privacy-Preserving Authorization and Low-Bandwidth Communication Devices." *IACR Cryptology ePrint Archive*, 2004, [GLS+04]

- L. Grassi, D. Khovratovich, and M. Schofnegger, "Poseidon2: A Faster Version of the Poseidon Hash Function." *IACR Cryptology ePrint Archive*, 2023, [GKS23]
- D. Kogan, "Lecture 5: Proofs of Knowledge, Schnorr's protocol, NIZK." *Stanford University*, 2019, [Kogan19]
- Blog post, "Notes and Proofs for Divisor Techniques": https://blog.zksecurity.xyz/posts/divisor-notes
- `polymesh-dart/dart-bp/docs`: https://github.com/PolymeshAssociation/polymesh-dart/tree/zk_audit_july_2026/dart-bp/docs
- P-DART: Polymesh's extension of DART using Bulletproofs (Technical Specification): https://polymeshassociation.github.io/polymesh-dart/dart-bp-specification.pdf
- Generalized Bulletproofs for Opening Vector Commitments: https://github.com/monero-oxide/monero-oxide/blob/fcmp%2B%2B/audits/generalized-bulletproofs/Fixed%20Protocol.pdf
- Discrete Logarithm Proofs Premised on Elliptic Curve Divisors: https://github.com/monero-oxide/monero-oxide/blob/fcmp%2B%2B/audits/divisors/README.md
- Standard curve database | Pallas: https://std.neuromancer.sk/other/Pallas
- Standard curve database | Vesta: https://std.neuromancer.sk/other/Vesta
- `snarkpack`: https://github.com/nikkolasg/snarkpack
- Module `discrete_log`: https://docs.rs/schnorr_pok/latest/schnorr_pok/discrete_log/
- Short Weierstrass curves: https://www.hyperelliptic.org/EFD/g1p/auto-shortw.html
- XYZZ coordinates for short Weierstrass curves: https://www.hyperelliptic.org/EFD/g1p/auto-shortw-xyzz.html
- Trait - Drop in `std::ops`: https://doc.rust-lang.org/std/ops/trait.Drop.html#panics

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Vulnerabilities within each component and whether the interaction between the components is secure;
- Whether requests are passed correctly to the network core;
- Key management, including secure private key storage and management of encryption and signing keys;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution;
- Protection against malicious attacks and other ways to exploit;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Polymesh DART is a Rust implementation of the Decentralized, Anonymous, and Regulation-friendly Tokenization protocol for confidential assets on the Polymesh blockchain, providing the cryptographic foundations for confidential asset transfers while supporting regulatory oversight through designated auditors and mediators.

The system combines encrypted transaction data, confidential account states, curve-tree membership proofs, generalized Bulletproofs, and linked Sigma protocols to prove that transactions are valid without revealing their underlying values.

The audit focused primarily on the `dart-bp` crate, particularly settlement, leg creation, affirmation, and split proofs. We traced prover-controlled inputs through constructors, circuit constraints, Sigma equations, sanitation checks, and verification paths. This covered encrypted amounts and asset identifiers, auditor and mediator keys, sender and receiver visibility, account authentication, balance conservation, range enforcement, and the binding between device and host proof contributions. We examined the code for missing constraints, unconstrained optional fields, arithmetic boundary errors and replay opportunities. Account registration, key distribution, fees, minting, state transitions, and Poseidon2 integration were reviewed as supporting parts of the confidential asset lifecycle. During our analysis, we identified an issue with split-device requests (W3) that enables a malicious host to spoof the operation being authorized by the device (Issue C). We also identified an issue with cross-party visibility that allows unauthenticated one-way party visibility (Issue B). Furthermore, due to a difference in how a prover-provided value is encoded and decoded on different architectures, validators may enter a consensus split, which could potentially affect finality (Issue D).

We also defined a set of protocol and implementation properties. For the settlement lifecycle, we assessed, to the extent supported by our analysis, whether:

- nullifiers remain unique;
- value is conserved and no balance becomes negative;
- a receiver is credited only after the sender has affirmed;
- a reversion restores exactly what was debited;
- a confirmed settlement cannot be reverted or re-affirmed;
- accumulator leaves are never removed;
- accounts stay unique per key and asset; and
- fee and main account state cannot be substituted for one another.

We checked these properties across all reachable states of the lifecycle, within bounds, and did not identify any cases that violate them. The bounds are two parties, one main asset and one fee asset, and a single settlement. Transfer amounts range up to two and balances up to five. The longest transition sequence explored is nine steps for the `dart-bp` model and 11 for the abstract protocol model.

Against an active network attacker who can corrupt parties, we identified no violations of the properties that settlement requires the consent of every party, that affirmations cannot be forged or replayed, and that leg contents remain confidential. We found no inconsistencies between the algebraic constructions of the `dart-bp` specification and those of the paper. We also identified no violations of these properties in the protocol models we used, which assume idealized cryptography. We did not establish a machine-checked link between those models and the Rust code.

At the implementation level, we encountered no cases in which account state operations panicked within bounds. The bounds are balances up to 10,000, counters up to 100, and transfer amounts up to 10,000.

This covers each operation individually and every sequence of two operations. We found no cases in which an on-chain proof type failed to reject malformed input or panicked during processing. The inputs consisted of random bytes sized for each proof type, ranging from 64 up to 5,000 bytes. The lifecycle invariants also hold at the implementation level. None of the property checks produced any findings.

Fiat–Shamir was analyzed across all in-scope NIZKs. We reviewed domain separation, statement and context absorption, challenge ordering and reuse, optional proof layouts, nonzero challenge assumptions, batch randomization, split-proof transcript binding, and binding to settlement and leg identifiers. The forked curve-tree components were reviewed for their local changes and security-critical use by DART. This included membership and rerandomization relations, range proofs, point commitments, `shared_dlog_indices`, divisor arithmetic, scalar decomposition, interpolation, discrete-log gadgets, and verifier coefficient coverage. The generalized Bulletproofs changes were compared with the referenced protocol and audit material, focusing on vector commitments, circuit equations, batch verification, transcript handling, and hash-to-curve behavior. The Schnorr, ElGamal, randomized verification, wrapper, compact point serialization, and modified multiscalar multiplication paths were also examined within their defined scope. We found that a missing check when indexing a fixed-length vector can trigger a denial of service due to a malformed proof (Issue A) and that the protocol transcripts lack enforced domain separation and complete context binding (Suggestion 6). Additionally, we identified an issue with the batch verifier implementation that allows malformed proof encodings to be accepted (Issue E). We also found that a malformed proof containing fewer responses than pending legs can panic the verifier (Issue F).

To understand the system and determine whether proof-layer behavior was exploitable in practice, we also examined adjacent components outside the defined scope. We traced proof outputs into the ledger lifecycle, including stored-leg binding, nullifier freshness, supply caps, account uniqueness, asset-key lookup, and settlement status and finality. We reviewed the testing CLI and mock-chain mediator flows to distinguish production issues from testing-only behavior. We also compared implementation assumptions with the P-DART paper, upstream protocol specifications, and prior divisor and generalized Bulletproofs audits. In addition, we examined other curve configurations and unused narrow-divisor paths beyond the production Pallas and Vesta path.

Overall, our analysis of the system's design and implementation revealed that security has been taken into consideration, as demonstrated by well-written code and securely designed circuits.

Dependencies

The repositories in scope include dependencies with known security issues. Running `cargo audit` identified three vulnerable crates: `tracing-subscriber`, `crossbeam-epoch`, and `rsa`. We recommend updating these dependencies where possible and improving dependency management (Issue G).

Code Quality

We performed a manual review of the repositories in scope and found the code to be of high quality, well organized, and aligned with Rust development best practices.

Tests

The project includes a test suite consisting of 184 tests, achieving 89.15% line coverage, 91.55% region coverage, and 71.07% function coverage of `dart-bp`. We observed the use of both positive and negative testing. However, extending the negative tests to mutate structural fields, such as index maps and length counts, could have identified Issue A and Issue F (Suggestion 5).

Documentation and Code Comments

The project documentation clearly outlines the project's purpose and sufficiently describes the system's intended functionality. Additionally, the codebase includes descriptive comments that, when considered alongside the documentation, sufficiently explain the behavior of the relevant components.

Scope

The review of adjacent components identified several issues in security-relevant code outside the defined scope, such as soundness failures, which have been also shared with the Polymesh team. We therefore recommend conducting an additional audit covering a broader portion of the codebase.

Specific Issues & Suggestions

We list the issues and suggestions identified during the review in descending order of severity. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	SEVERITY	STATUS
Issue A: Malformed Partial Schnorr Response Causes a Panic During Split Affirmation Verification	High	Resolved
Issue B: Asymmetric Visibility Fields Bypass Cross-Party Proof Constraints	Medium	Resolved
Issue C: Opaque Split-Device Requests Allow Operation Spoofing by a Malicious Host	Medium	Resolved
Issue D: Architecture-Dependent Index Decoding Can Split Hidden-Leg Verification	Medium	Resolved
Issue E: Constant Batching Coefficient Allows Malformed Proof Encodings to Cancel	Low	Resolved
Issue F: Missing Participant-Response Length Check Can Panic Proof-of-Balance Verification	Low	Resolved
Issue G: Usage of Vulnerable Dependencies	Undetermined	Resolved
Suggestion 1: Enforce Cryptographically Secure Randomness for Randomized Batch Verification	Informational	Resolved
Suggestion 2: Validate the Declared Leg Count Against the Chunk Contents	Informational	Resolved
Suggestion 3: Reject Unused r4 Fields From Revealed-Asset Legs	Informational	Resolved
Suggestion 4: Bind Each Mediator p_1 Transcript to the Point Used by Its Verification Equation	Informational	Resolved

Suggestion 5: Extend Negative Testing	Informational	Partially Resolved
Suggestion 6: Strengthen Domain Separation and Context Binding in Protocol Transcripts	Informational	Resolved

Issue A: Malformed Partial Schnorr Response Causes a Panic During Split Affirmation Verification

Location

schnorr_pok/src/partial.rs#L229-L235

schnorr_pok/src/partial.rs#L32-L38

utils/src/randomized_mult_checker.rs#L379-L397

utils/src/randomized_mult_checker.rs#L132-L145

src/bp/affirmation_proofs.rs#L307-L308

dart-bp/src/auth_proofs/helpers.rs#L108-L120

worker/native/src/lib.rs#L48-L72

Synopsis

The `PartialSchnorrResponse` structure carries a map of witness indices that the submitted proof supplies. The verifier `pre_verify` function checks the size of this map but does not check the individual index values. It then writes each value into a fixed-length vector with an unchecked index operation. An index above the vector length makes the verifier panic instead of returning an error. An attacker who submits one malformed split affirmation proof stops verification and causes a denial of service.

Preconditions

The attacker must hold an account that is an authorized party on a pending settlement leg (sender or receiver). This is a standard operational role.

Likelihood

High.

The attacker must register an account for the asset, create a settlement that names its own accounts, and then affirm a leg of that settlement.

The attacker pays no transaction fee because the node aborts before it commits the block. The attack therefore incurs no cost beyond a one-time account registration, and the attacker can repeat it without limit.

Impact

Medium.

The verifier panics during proof verification, leading to a loss of availability of the verification functionality.

The split affirmation path always runs the randomized multiplication checker, and a second panic follows during unwinding. Rust aborts the process on that second panic. The worker wraps verification in `catch_unwind`, but an abort is not an unwind, so that containment does not apply. The node becomes unavailable, causing every user of that node to lose service. Because the attack incurs no additional cost and is repeatable, a restart returns the node to service only until the attacker sends the next proof.

Severity

High.

Technical Details

`PartialSchnorrResponse` holds a public map named `responses` and a count named `total_responses`. The map key is a witness index. The structure derives `CanonicalDeserialize`, and the keys are therefore derived from the submitted proof bytes.

Split affirmation verification supplies three bases for the updated account commitment, which are the secret key generator, the encryption key generator, and the commitment rerandomization generator. It provides the responses for witness index `0` and witness index 1, the two key generators, as the missing responses. The proof supplies the single remaining response, witness index 2.

`pre_verify` allocates a vector of length three and applies four checks. It confirms that `total_responses` equals the number of bases, that the two maps together contain the corresponding number of entries, that the maps are disjoint, and that each missing response key is in range. Line 230 enforces the last check. Line 235 then writes the entries from the `responses` map with the statement `full_resp[*i] = *r`. No check applies to `i`.

The attacker modifies a valid split affirmation proof by replacing map key 2 with key 3. It leaves `total_responses` at 3 and every other field unchanged. All four checks still pass. Line 235 writes to element 3 of a vector of length 3, causing the indexing operation to panic before any cryptographic check runs. The panic results in an abort. The split affirmation verifier builds a `PairRandomizedMultCheckerGuard` and calls its `with` method, which passes a live checker into verification. That method cancels both checkers on the error branch of its match, but a panic unwinds straight past the match. Each checker reaches its destructor at line 379 with the `cancelled` flag and the `verified` flag both false. The destructor treats that state as a programming fault and verifies the accumulated equations. The check fails only when an equation already in the accumulator fails, because an empty accumulator returns success and a set of valid equations sums to zero. The attacker therefore provides an invalid response for the rerandomized account commitment, which the checker defers rather than rejects. The destructor then encounters the invalid equation and panics. Rust aborts the process if a panic occurs during unwinding.

Remediation

We recommend replacing the unchecked index at `partial.rs` line 235 with checked access and requiring the disjoint union of both key sets to be exactly the range `0` to `total_responses`:

```rust
full_resp.get_mut(*i).ok_or(SchnorrError::IndexOutOfBounds(*i, n))?
```

As an additional measure, the arkworks `Valid` check can be implemented for `PartialSchnorrResponse` so that a malformed proof fails at deserialization and never reaches the verifier.

Separately, we recommend preventing the `RandomizedMultChecker` destructor from panicking while the thread is already unwinding. Returning early in that case removes the abort:

```rust
#[cfg(feature = "std")]
if std::thread::panicking() {
       log::error!("RandomizedMultChecker dropped during unwind without verify()");
       return;
}
```

Status

The Polymesh team confirmed that panics are captured by Substrate and that proofs that panic are considered invalid.

Verification

Resolved.

Issue B: Asymmetric Visibility Fields Bypass Cross-Party Proof Constraints

Location

dart-bp/src/leg/leg_proof.rs#L1179-L1200

dart-bp/src/leg/public_asset_leg_proof.rs#L679-L703

Synopsis

The leg verifier treats counterparties as visible only when both optional cross-party fields are present. If exactly one field is present, the verifier disables both cross-party proof checks but retains the lone field, allowing an undocumented and unauthenticated one-way visibility state.

Preconditions

The attacker must control leg creation and construct the encrypted leg without relying on the honest `LegEncConfig` constructor. All other witnesses and proof components must be valid, and no outer integration may reject asymmetric field presence.

Likelihood

High.

A malicious leg creator can select the asymmetric fields before generating a fresh proof. Exploitation does not require modifying an existing proof, reusing a challenge, or breaking a cryptographic assumption.

Impact

Low.

The issue violates the documented mutual-visibility policy and can disclose one counterparty key unilaterally or introduce an unauthenticated cross-party point. It does not bypass amount consistency, balance conservation, asset binding, or account ownership constraints.

Severity

Medium.

Technical Details

Mutual visibility is computed as `eph_pk_s.r2.is_some() & eph_pk_r.r1.is_some()`. Both asymmetric states therefore evaluate to false. The verifier then requires both cross-party responses to be absent but does not require both statement fields to be absent. The corresponding Sigma equations and circuit constraints are also disabled, while sanitation only checks that present points are nonidentity and distinct from other ephemeral points.

The malformed leg is included in the Fiat–Shamir transcript, which prevents later modification but does not establish the missing relation. The attacker can choose one cross-party point, omit the other point and both responses, then generate a fresh proof that matches this layout.

Remediation

We recommend enforcing paired presence in both verifiers if visibility must remain bilateral. If one-way visibility is supported, all four visibility states should be represented explicitly, with the corresponding Sigma response and constraints required independently for each present field.

Status

The Polymesh team has resolved this issue as recommended.

Verification

Resolved.

Issue C: Opaque Split-Device Requests Allow Operation Spoofing by a Malicious Host

Location

`src/bp/fee_split.rs#L66-L105`

`src/bp/fee_split.rs#L146-L210`

`src/bp/account_reg_split.rs#L26-L74`

`src/bp/mint_split.rs#L47-L115`

`dart-bp/docs/1.md#L328-L341`

Synopsis

Split-device requests (W3) do not identify the operation or expose its parameters to the trusted device. A malicious host can therefore present a request as registration while obtaining a valid device authorization for minting or fee top-up. The proof remains cryptographically bound to the host's actual statement. The vulnerability is semantic pre-authorization spoofing, not replay or post-signing modification.

Preconditions

The split-device request (W3) workflow must be used. In addition, the host must be malicious or compromised, and the device must be intended to provide independent user authorization.

Likelihood

Medium.

Compromising the host is a meaningful prerequisite. However, it remains part of the threat model, and constructing the undisclosed request requires no cryptographic bypass.

Impact

Medium.

The device may authorize minting or fee top-up without the user knowingly approving that operation or its parameters. Existing verifier constraints, authorization rules, balance limits, and supply caps continue to apply.

Severity

Medium.

Technical Details

When running the sequential split-device workflow (W3), an untrusted host requests authorization from a trusted device that stores the user's secret keys. `FeeAccountDeviceRequest` is shared by fee-account registration and fee top-up, while `RegistrationDeviceRequest` is shared by account registration and minting:

```rust
FeeAccountDeviceRequest {
    challenge_h_bytes,
    nonce,
    pk,
}

RegistrationDeviceRequest {
    challenge_h_bytes,
    nonce,
    pk_aff,
    pk_enc,
}
```

Neither request contains an operation identifier, asset ID, or amount. Although `challenge_h_bytes` cryptographically binds the authorization to the host's actual statement, it is opaque to the device and cannot be used to independently identify or display that operation. The device constructs its authorization using this opaque challenge:

```rust
let nonce = ledger_nonce(&request.challenge_h_bytes, &request.nonce);
let proof = BPAuthProofOnlySk::new(..., &nonce, ...)?;
```

Consequently, a malicious host can prepare a mint or top-up statement while presenting the request to the user as account or fee-account registration. The device produces a valid authorization for the undisclosed statement, which the verifier correctly accepts.

Remediation

We recommend canonically encoding and binding the complete operation and parameters into the device transcript. The device should parse and require confirmation of the expected operation and parameters from the transaction being verified, rather than trusting an operation identifier supplied only by the host. Separate transcript domains or proof types should be used for each operation while retaining the existing binding to the host challenge `challenge_h_bytes`. Additionally, typed, operation-specific device requests containing the semantic fields required for approval should be introduced:

```rust
enum DeviceOperation {
    AccountRegistration { asset_id: AssetId },
    Mint { asset_id: AssetId, amount: Balance },
    FeeRegistration { asset_id: AssetId },
    FeeTopUp { asset_id: AssetId, amount: Balance },
}
```

Status

The Polymesh team has resolved the issue as recommended.

Verification

Resolved.

Issue D: Architecture-Dependent Index Decoding Can Split Hidden-Leg Verification

Location

`relations/src/ped_comm_group_elems.rs#L81`

`relations/src/ped_comm_group_elems.rs#L283-L284`

`serialize/src/impls/int_like.rs#L152`

Synopsis

Hidden-leg proofs contain a prover-controlled map whose indices are encoded as `u64` but decoded into platform-width `usize` values using an unchecked cast. The same proof bytes can therefore produce different map indices under `wasm32` and `64-bit` native execution. If the same consensus transition can execute through both paths, validators may disagree on whether a settlement proof is valid.

Preconditions

For this issue to be exploitable, the following conditions must be met:

- The attacker must submit an otherwise valid hidden-asset leg.
- The asset must have at least one auditor encryption key or mediator key.
- The attacker must replace an expected index `k` with `2^32 + k`.
- Consensus execution must be able to use both `wasm32` and `64-bit` native runtime semantics for the same transition.

Likelihood

Low.

Exploitation requires mixed consensus execution, where the same transition may run under both `wasm32` and `64-bit` native semantics.

Impact

High.

Affected validators can disagree on transaction validity, dispatch results, and resulting state roots. This could split consensus or interrupt finality.

Severity

Medium.

Technical Details

The `blindings_with_different_gen` map is used to store secondary commitments to the blindings used to rerandomize hidden auditor encryption keys and mediator affirmation keys:

```rust
pub blindings_with_different_gen: BTreeMap<usize, Affine<P>>
```

The map is deserialized from prover-controlled proof bytes, and its indices identify the rerandomized asset-leaf points to which the commitments correspond. The verifier constructs the expected small index set and requires the decoded map to match it:

```rust
shared_dlog_indices.iter().all(|i| {
    re_randomized_points
        .blindings_with_different_gen
        .contains_key(i)
        && *i < size
})
```

arkworks serializes values of type `usize` (which is used for the map indices) as `CompactU64`, but deserializes them using an unchecked target-width conversion:

```rust
let len = CompactU64::deserialize_with_mode(
    &mut reader,
    compress,
    validate,
)?.0;
Ok(len as usize)
```

On a `32-bit` target such as `wasm32`, values greater than `2^32-1` are silently truncated. For a serialized key `2^32 + k`:

- `wasm32` decodes the value as `k`. The expected-key check passes, and the Fiat–Shamir transcript remains valid because it consumes the decoded map, which is identical to the honest map.
- A `64-bit` native build retains `2^32 + k`, causing the verifier to return `MalformedProofInput`.

If consensus execution can use both targets for the same transition, validators may disagree on whether the proof is valid.

Remediation

We recommend replacing serialized `usize` indices with a fixed-width semantic type such as u32 or u64 instead of relying on platform-dependent types. Values outside the protocol's permitted index range should be rejected identically on every target.

Status

The Polymesh team has resolved the issue as recommended.

Verification

Resolved.

Issue E: Constant Batching Coefficient Allows Malformed Proof Encodings to Cancel

Location

`bulletproofs/src/r1cs/verifier/batch.rs#L32`

Synopsis

The Bulletproof batch verifier applies the same randomly generated coefficient to multiple verification tuples. Consequently, two individually invalid proof encodings with equal and opposite verification residuals can cancel and be accepted as a batch. The demonstrated construction malleates two copies of an otherwise valid statement. It does not, by itself, demonstrate acceptance of a false balance, unauthorized operation, or other semantically false statement.

Preconditions

The attacker must control at least two tuples assigned the same batching coefficient. The tuples must have matching Fiat–Shamir transcripts so their residuals can be constructed as opposites. Additionally, for the same-size branch, the first tuple must be valid, followed by at least two canceling malformed tuples.

Likelihood

Low.

The cancellation is deterministic. However, exploitation requires an external production caller using the affected batching API and permitting multiple tuples with identical transcripts.

Impact

Medium.

A successful attack violates the batch verifier's guarantee that every accepted tuple is individually valid. This can create inconsistent results between batch and standalone verification and could affect integrity or availability if later components rely on individual proof validity. The demonstrated attack only produces

malformed encodings of a statement the attacker can already prove. No false protocol statement, unauthorized state transition, or loss of funds has been demonstrated.

Severity

Low.

Technical Details

The `batch_verify_with_rng` function samples a nonzero scalar but passes a closure that ignores its input:

```rust
let mut r = C::ScalarField::rand(rng);
while r.is_zero() {
    r = C::ScalarField::rand(rng);
}
batch_verify_core(verification_tuples, pc_gens, bp_gens, |_| r)
```

Consequently, every tuple after the first receives the same coefficient in the same-size branch. In the different-size branch, every tuple receives that coefficient. The inner-product proof's a value is read after its transcript challenges are generated. As a result, an attacker can clone a valid proof and modify the two copies in opposite directions:

```rust
forged_plus.ipp_proof.a += delta;
forged_minus.ipp_proof.a -= delta;
```

Each encoding fails standalone verification, but their opposite errors cancel when both receive the same batching coefficient.
The sibling implementation, `batch_verify_with_given_randomness`, preserves the accumulator, which produces distinct successive powers and rejects the malformed pair:

```rust
batch_verify_core(
    verification_tuples,
    pc_gens,
    bp_gens,
    |previous| randomness * previous,
)
```

Remediation

We recommend assigning a distinct unpredictable coefficient to every verification tuple. Alternatively, successive powers of one random nonzero coefficient can be applied by changing the closure to preserve its accumulator:

```rust
batch_verify_core(verification_tuples, pc_gens, bp_gens, move |previous| previous
* r)
```

Preferably, an independent nonzero coefficient should be sampled for each tuple after the complete batch has been received. The same-size and different-size branches should follow the same coefficient convention.

Status

The Polymesh team has resolved the issue as recommended.

Verification

Resolved.

Issue F: Missing Participant-Response Length Check Can Panic Proof-of-Balance Verification

Location

`dart-bp/src/account/pob.rs#L664`

`dart-bp/src/account/pob.rs#L855`

`dart-bp/src/account/pob.rs#L861`

`dart-bp/src/account/pob.rs#L964`

`dart-bp/src/account/pob.rs#L977`

Synopsis

The `PobWithAnyoneProof::verify_with_given_transcript` function indexes the proof-controlled `resp_participant` vector without verifying that its length matches the number of pending legs. A malformed proof containing fewer responses can therefore panic the verifier.

Preconditions

An external application must expose `PobWithAnyoneProof` verification to untrusted users. In addition, the verifier must supply at least one pending leg, and the submitted proof must contain fewer `resp_participant` entries than legs.

Likelihood

Medium.

The malformed input is simple to construct and does not require a valid cryptographic proof. Exploitation nevertheless depends on an external application exposing this lower-level verifier.

Impact

Low.

Repeated panics could terminate or restart an off-chain verification service, preventing legitimate proof-of-balance requests from being processed. Since Proof of Balance is an off-chain verifier, the impact is considered low.

Severity

Low.

Technical Details

The verifier validates the length of `resp_asset_id`:

```rust
if self.resp_asset_id.len() != legs.len() {
    return Err(Error::ProofVerificationError(...));
}
```

However, it performs no equivalent check for `resp_participant`. Instead, it uses the number of legs as the loop bound:

```rust
let num_pending_txns = legs.len();
for i in 0..num_pending_txns {
    self.resp_participant[i].challenge_contribution(...)?;
}
```

When `resp_participant.len() < legs.len()`, indexing causes an out-of-bounds panic before the proof is rejected. The vector is indexed again during response verification.

Remediation

We recommend performing a length check on `resp_participant` before transcript construction and indexing, returning an error if the length does not match the number of legs:

```rust
if self.resp_participant.len() != legs.len() {
    return Err(Error::ProofVerificationError(format!(
        "resp_participant length mismatch: expected {}, got {}",
        legs.len(),
        self.resp_participant.len(),
    )));
}
```

Additionally, regression tests should be included using empty and truncated response vectors with nonempty leg lists.

Status

The Polymesh team has resolved this issue in PR 12.

Verification

Resolved.

Issue G: Usage of Vulnerable Dependencies

Synopsis

Analysis of the project's dependencies using `cargo audit` revealed three vulnerable crates:

- `tracing-subscriber v0.2.25`: Logging user-controlled input may allow ANSI escape sequence injection and log poisoning, as described in this security advisory.

- `crossbeam-epoch v0.9.18`: Formatting a `null` or otherwise invalid pointer of type `Atomic` or `Shared` can cause an invalid memory access due to dereferencing, as described in this security advisory.
- `rsa v0.9.10`: Recovering keys through timing side channels may be possible through the Marvin attack, as described in this security advisory.

Likelihood
Consult the listed advisories on a case-by-case basis.

Impact
Consult the listed advisories on a case-by-case basis.

Severity
Consult the listed advisories on a case-by-case basis.

Technical Details
The Polymesh DART implementation includes dependencies with known security issues that have since been resolved in updated versions.

Remediation
We recommend updating these dependencies and following a process that emphasizes secure dependency usage to avoid introducing vulnerabilities into the Polymesh DART implementation and to mitigate supply-chain attacks. This process includes:

- Manually reviewing and assessing currently used dependencies;
- Upgrading dependencies with known vulnerabilities to patched versions with fixes;
- Replacing unmaintained dependencies with secure and battle-tested alternatives, if possible;
- Pinning dependencies to specific versions, including pinning build-level crates;
- Only upgrading dependencies upon careful internal review for potential backward compatibility issues and vulnerabilities; and
- Including automated dependency auditing reports in the project's CI/CD workflow.

Status
The Polymesh team has resolved the issue as recommended by updating the `crossbeam-epoch` crate to a more recent version. Both `tracing-subscriber` and `rsa` remain at vulnerable versions, as the Polymesh team stated that they are not used in P-DART.

Verification
Resolved.

Suggestions

Suggestion 1: Enforce Cryptographically Secure Randomness for Randomized Batch Verification

Location
PolymeshAssociation-crypto/utils/src/randomized_mult_checker.rs#L46

PolymeshAssociation-crypto/utils/src/randomized_mult_checker.rs#L121

Synopsis

The randomized batch checkers accept any Rng, although their soundness requires fresh and unpredictable verifier randomness. The type signatures therefore permit accidental use of deterministic or noncryptographic generators.

Mitigation

We recommend requiring `RngCore + CryptoRng` for constructors that sample batching coefficients. Current verifiers should supply a properly seeded cryptographic RNG.

Status

The Polymesh team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 2: Validate the Declared Leg Count Against the Chunk Contents

Location

`PolymeshAssociation-polymesh-dart/dart-bp/src/leg/settlement_proof_chunked.rs#L44`

Synopsis

The chunked verifier compares the supplied leg encryptions with the sum of the actual chunk lengths but does not require `num_legs` to equal that sum. The declared cardinality can therefore disagree with the proof contents, creating ambiguous or malleable metadata that may become security relevant if downstream code trusts it.

Mitigation

We recommend computing the total number of contained leg proofs using checked arithmetic and requiring both `num_legs` and the supplied leg-encryption count to equal it. The proof should be rejected when either value differs.

Status

The Polymesh team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 3: Reject Unused r4 Fields From Revealed-Asset Legs

Location

`PolymeshAssociation-polymesh-dart/dart-bp/src/leg/public_asset_leg_proof.rs#L625-L675`

Synopsis

Leg verification for a revealed asset accepts optional `r4` points even though the public proof contains no relation involving them. Each accepted `r4` is therefore an unconstrained statement field that the prover

controls and whose only binding is Fiat–Shamir. Leg creators could therefore encode arbitrary data into valid curve points and use these fields as covert memo or data-upload channels.

Mitigation

We recommend requiring every sender, receiver, auditor, and public-encryption-key `r4` field to be absent when the asset ID is revealed. The proof should be rejected before processing its remaining relations if any such field is present.

Status

The Polymesh team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 4: Bind Each Mediator p_1 Transcript to the Point Used by Its Verification Equation

Location

PolymeshAssociation-polymesh-dart/dart-bp/src/leg/leg_proof.rs#L794

PolymeshAssociation-polymesh-dart/dart-bp/src/leg/leg_proof.rs#L1472

PolymeshAssociation-polymesh-dart/dart-bp/src/leg/leg_proof.rs#L1785

Synopsis

The mediator `p_1` equation is verified against the secondary point `B * k_i`, while its challenge contribution uses the primary rerandomized mediator point. Consequently, the isolated Sigma transcript does not directly bind the statement used by its verification equation. The point-commitment proof independently binds both points to the same scalar, preventing an exploitable soundness issue. However, aligning the transcript and equation would make the construction internally consistent and simplify its security argument.

Mitigation

We recommend using `blindings_with_different_gen[1 + n + i]` in the mediator `p_1` challenge contribution so that the transcript serializes the same secondary point used by the verification equation.

Status

The Polymesh team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 5: Extend Negative Testing

Synopsis

The test suite applies negative testing consistently but is narrow. Each test builds a well-formed proof, replaces a witness value with an incorrect value, and requires verification to fail. The locations above cover registration, minting, transparent withdrawals, and split affirmations. These tests exercise the cryptographic checks well.

A second class of input remains uncovered. A proof can decode successfully, carry well-formed cryptographic material, and still be malformed in its structure. Such input reaches verifier code before any cryptographic check runs.

Two of the issues in this report fall in that class. First, the verifier receives a response vector shorter than the leg count and indexes it (Issue F). In addition, the verifier receives a map key above the response count and writes through it (Issue A).

Additionally, line coverage of the error paths reflects the gap (33% for `error.rs` and 76% for `account/common/verifier.rs`).

Mitigation

We recommend broadening the negative tests to cover malformed but decodable input. Note that fuzzing would not reach these cases either. Random bytes almost never decode into a structurally valid proof that carries a single out-of-range field.

`PobWithAnyoneProof` provides a representative example. It holds two vectors that the proof controls, `resp_participant` and `resp_asset_id`. The prover builds both in the same loop, and the verifier indexes both. The verifier checks the length of `resp_asset_id` against the leg count but applies no such check to `resp_participant`. The suite already tests the checked field. The test `pob_with_anyone_resp_asset_id_len_mismatch_fails` removes one entry from `resp_asset_id` and requires an error. Applying that same test to `resp_participant` yields Issue F.

To identify valuable negative testing targets, we recommend listing every caller-controlled value that a verifier consumes. For each value, the assumptions made by the verifier should be identified, along with where the code enforces them. A test should be added wherever the enforcement is missing.

The following areas have already produced findings in this review and could be used as starting points:

- Lengths and counts that must match a verifier-side value.
- Indices and map keys that must remain within a bound.
- Optional fields whose absence disables a check.
- Boundary values, such as zero amounts, zero counters, and empty vectors.
- Specific errors that should be asserted because a test that accepts any failure still passes after a change causes the incorrect check to occur first.

Status

The Polymesh team has added negative tests for the two issues referenced in this suggestion but has not carried out the recommended thorough negative testing of verifier inputs. As a result, the class of issues that rely on correctly decodable but malformed inputs remains undertested.

Verification

Partially Resolved.

Suggestion 6: Strengthen Domain Separation and Context Binding in Protocol Transcripts

Location

PolymeshAssociation-crypto/schnorr_pok/src/discrete_log.rs#L117-L153

PolymeshAssociation-crypto/schnorr_pok/src/pok_generalized_pedersen.rs#L124-L125

PolymeshAssociation-crypto/schnorr_pok/src/mult_relations.rs#L170-L173

PolymeshAssociation-crypto/schnorr_pok/src/discrete_log_pairing.rs#L98-L106

Synopsis

Challenge-contribution APIs serialize proof elements without intrinsically binding a protocol identifier, version, relation tag, or complete application context. Callers must therefore provide sufficient external framing to prevent transcript collisions across protocol roles. In the in-scope P-DART implementation, higher-level values are bound, and we did not identify an exploitable path. However, stronger API-level domain separation would reduce the risk of replay, substitution, or type confusion in future uses.

Mitigation

We suggest that users construct each Fiat–Shamir transcript with a unique domain separator and bind the full statement, protocol role, and relevant application or transaction context. Distinct relation tags should be used for semantically different proofs, particularly when the same low-level APIs are reused across protocol roles.

Status

The Polymesh team has strengthened domain separation and context binding in P-DART, including rejecting empty separators and tests for duplicate DART relation tags. The underlying cryptographic APIs retain flexible transcript construction and do not universally enforce a nonempty domain separator.

Verification

Resolved.

About Least Authority

Founded in 2011 and based in Berlin, Germany, since 2016, Least Authority provides security consulting for privacy-preserving technologies and decentralized systems. We are committed to transparency, openness, and advancing secure digital infrastructure.

We believe that people have a fundamental right to privacy and that secure solutions enable people to use the internet and other connected technologies more freely. Our security consulting services help teams make their solutions more resistant to unauthorized access and unintended system manipulation. We support teams from the design phase through production launch and beyond.

Our team has experience reviewing code for security vulnerabilities and specific attack vectors in multiple languages, including Rust, Go, Python, C, C++, Solidity, Cairo, Circom, Noir, Haskell, OCaml, TypeScript, Swift, ZoKrates, and others. We have reviewed implementations of cryptographic protocols and distributed system architecture in blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, we use various tools to scan code and networks and build custom tools as necessary.

For more information about our security consulting, please visit:
https://leastauthority.com/security-consulting.

Our Methodology

We use a transparent review process that provides the project team visibility into our findings, assumptions, and remediation considerations throughout the audit. The goals of our security audits are to improve the quality of the systems we review and support sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Code Review

In manually reviewing all of the code, we identify potential issues, including those related to code logic, error handling, cryptographic implementations, and correctness with respect to specifications, standards, and best practices. We also evaluate areas where more defensive programming could reduce the risk of future mistakes and accelerate future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

When appropriate, we also use artificial intelligence (AI) to support the efficiency of our security audits without reducing rigor or reliability. AI-assisted outputs are carefully guided, reviewed, and validated by our team so that conclusions remain accurate and defensible. This supports stronger audit decisions, not only faster review.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and white box penetration testing, supplemented by various tools, including AI, that support the code review process. Discussions with the development team help clarify their vision for the software. Installing and using the relevant software helps us understand user interactions and roles, while also informing our analysis of threat models and attack surfaces. Design documentation, relevant papers, other audit results, similar projects, source code dependencies, and open issue tickets may also be reviewed to inform the analysis. We

identify potential vulnerabilities, and, when appropriate, create issue entries. For each entry, we follow the issue investigation and remediation process described below.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and supporting successful remediation. Whenever a potential issue is identified, we raise it with the client through a dedicated secure communication channel to discuss its validity and impact, to the extent these can be determined at that stage. This process allows potentially relevant findings to be reviewed with the client early, before they are documented in greater detail in the initial audit report. For each confirmed issue, we assess impact and feasibility and use those ratings to determine the overall severity, following the OWASP Risk Severity Matrix.

Suggested Solutions

We identify immediate and comprehensive mitigations available to live deployments and then suggest remediation requirements for future releases. When code is already deployed in production, we include mitigation strategies in the report to provide interim support until appropriate remediation is available. The mitigation and remediation recommendations should be reviewed by the development team and deployment engineers. After delivering the Initial Audit Report, we remain available to support implementation of the proposed mitigations and remediations before the verification review.

Development teams should address findings in an order and with an urgency that prioritizes the security of their users. Findings should be addressed as soon as possible, but remediation should be carried out pragmatically and should not cause unnecessary disruption to project timelines.

Although issues must be handled on a case-by-case basis, we work toward a resolution timeline that balances the impact on users and the needs of the project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess whether the issues and suggestions have been sufficiently addressed. When this analysis is complete, we update the report and provide a Final Audit Report that can be published in full. If any critical issues remain, we recommend that the Final Audit Report not be published and that users and other stakeholders be alerted to the impact. We encourage all findings to be addressed and the Final Audit Report to be shared publicly to support transparency and advance security knowledge within the industry.