

Privacy Contracts Audit



May 29, 2026

This document has been prepared solely for the named client and it may not be relied upon by any third parties. The content in this document is provided for informational purposes only and does not constitute professional advice of any kind. Security assessments are limited by factors such as scope, time, and techniques used and therefore this document does not constitute a certification, guarantee, or warranty regarding the proper functioning of any system or the absence of security vulnerabilities.

Copyright © 2026 Zeppelin Group Ltd.

Table of Contents

Table of Contents	2
Summary	3
Scope	4
System Overview	5
Security Model and Trust Assumptions	6
Privileged Roles	6
Medium Severity	7
M-01 Invalid Auditor Public Keys Can Brick Registration And Withdrawals	7
M-02 Auditor Key Rotation Breaks Continuous Audit Visibility Across Channels	7
Low Severity	8
L-01 Signature Validation Occurs After Expensive Compilation in __execute__	8
L-02 Nominal Token Accounting Breaks on Fee-On-Transfer and Rebasing Tokens	9
L-03 Invalid Public Keys Trigger Generic Panic Instead of Explicit Error	9
L-04 pack Does Not Enforce 120-bit Constraint on value_1	10
L-05 Ephemeral Secret Reuse in ECDH Encryption Breaks Channel Privacy	10
Notes & Additional Information	11
N-01 Salt Type Inconsistency Across Encryption Hashes	11
N-02 Inconsistent Use of Reserved Slots in Hash Constructions	11
N-03 Comment Inconsistency on Viewing Key Mutability	12
N-04 Access Control Roles Marked as TODO in Sensitive Functions	12
Conclusion	13
Appendix	14
Issue Classification	14

Summary

Type	DeFi	Total Issues	11 (6 resolved)
Timeline	From 2026-03-18 To 2026-03-31	Critical Severity Issues	0 (0 resolved)
Languages	Cairo	High Severity Issues	0 (0 resolved)
		Medium Severity Issues	2 (2 resolved)
		Low Severity Issues	5 (1 resolved)
		Notes & Additional Information	4 (3 resolved)
		Client Reported Issues	0 (0 resolved)

Scope

OpenZeppelin audited the [starkware-libs/starknet-privacy](#) repository at the [c5e2fb5](#) commit.

In scope were the following files:

```
packages/privacy/src
├─ actions.cairo
├─ errors.cairo
├─ events.cairo
├─ hashes.cairo
├─ interface.cairo
├─ lib.cairo
├─ objects.cairo
├─ privacy.cairo
└─ utils.cairo
```

System Overview

The Starknet Privacy Pool is a UTXO-based privacy protocol designed as infrastructure for Starknet's STRK20 privacy standard. It enables private ERC-20 token transfers by allowing users to deposit into a shared pool, transact privately, and withdraw without revealing sender-recipient linkage.

The contract is deployed as a Starknet account contract (`#[starknet::contract(account)]`), exposing `__validate__` and `__execute__` for simulation and `apply_actions` for applying proven state transitions. A discovery service indexes encrypted on-chain data to allow clients to efficiently retrieve relevant channels and notes. Importantly, `__execute__` should not be called on-chain as the private key is embedded inside of the calldata.

The protocol consists of three core components:

Channel system

The channel system enables private transactions between two users. It derives a shared `channel_key` from both parties' addresses and viewing keys and communicates it using ECDH over the Stark curve. Each channel can be split into subchannels, with each subchannel corresponding to a different token. The token also encrypted with other parameters to ensure privacy.

Note system

A note system represents value as discrete notes identified by `note_id` with encrypted amounts, and uses nullifiers derived from viewing keys for spending. There are two types of notes: encrypted and open notes. Encrypted notes pack a random salt and the note amount into a single storage field. The amount is masked by adding a Poseidon-based key stream derived from the channel key, token, note index, and salt, so only the recipient can recover the amount. Open notes are created with initially zero value and a token address, acting as placeholders for amounts that will only be determined at execution time such as the output of an AMM swap or funds arriving through a bridge transfer.

Proof System

A ZK proof system compiles user actions off-chain via a centralized proving service running a virtual Starknet OS (Virtual SNOS), which generates proofs verified by Starknet consensus before state transitions are applied. State changes are committed through a write-once storage system that asserts a slot is empty before writing to it.

Security Model and Trust Assumptions

- The protocol relies on off-chain components for execution, proof generation, and data access. While the contract enforces state transitions, any party capable of running the proving infrastructure can submit valid transactions.
- Users depend on access to a proving service to generate valid transactions, though this service can be operated by any party.
- An auditor holds a private key corresponding to `auditor_public_key` and can decrypt viewing keys and withdrawal data recorded by the protocol. This provides visibility into user activity but does not grant the ability to control or spend user funds.
- Governance can update parameters and upgrade the contract with no delay. Users have no on-chain reaction window to governance changes.
- The protocol depends on Starknet consensus for proof validation, on clients for correct randomness generation, and on off-chain discovery services for state access.
- The privacy of ECDH-encrypted channel data depends on the client generating a fresh ephemeral secret for each encryption event. The contract validates only that the value is non-zero.

The following roles have privileged access within the protocol.

Privileged Roles

Throughout the system, the following privileged roles have been identified:

- **Governance Admin:** manages roles and upgrades
- **Token Admin:** sets auditor public key
- **App Governor:** sets fees and proof parameters
- **Auditor:** decrypts user-related data
- **Proving Service:** executes actions and submits proofs

Medium Severity

M-01 Invalid Auditor Public Keys Can Brick Registration And Withdrawals

The protocol relies on a global auditor public key for ECDH-based encryption in `set_viewing_key`, `withdraw`, and `create_open_note`. This key can be set at deployment or updated via an admin function, but is only validated as non-zero. The ECDH helper reconstructs a curve point using `EcPointTrait::new_from_x(...).unwrap()`, which panics if the x-coordinate does not correspond to a valid Stark curve point. As a result, an invalid but non-zero auditor key can be stored.

If a malformed auditor key is set, all flows that depend on auditor encryption revert, including user registration and withdrawals. This results in a system-wide liveness failure that blocks onboarding and fund exits until the key is corrected. If misconfigured at deployment, the protocol can be unusable from initialization and requires governance intervention (e.g., key update or upgrade) to recover.

Consider validating the auditor public key at the time it is set (both in the constructor and admin setter) to ensure it corresponds to a valid curve point. Additionally, replace the unchecked `unwrap()` with a checked path that returns a dedicated error.

Update: Resolved in [pull request #720](#) at commit [539f9e1](#).

M-02 Auditor Key Rotation Breaks Continuous Audit Visibility Across Channels

The auditor public key can be updated via `set_auditor_public_key`. Previously stored encrypted data including `enc_private_key` remains encrypted under the key active at the time of registration. Since channel keys are derived from user viewing keys, a new auditor cannot derive channel keys for users registered before rotation and cannot decrypt activity on those channels.

Under the intended compliance model where the auditor is expected to maintain continuous visibility, this creates a persistent audit gap. The new auditor cannot observe transactions on pre rotation channels even for activity occurring after rotation. Full audit coverage requires

access to both old and new auditor keys with no on-chain mechanism to enforce this. As a result users with pre rotation channels can continue transacting outside the visibility of the active auditor.

Consider documenting this limitation explicitly or introducing a migration process to preserve audit continuity across rotations. For example, user viewing keys could be re-encrypted to the new auditor prior to rotation via `set_auditor_public_key`, or channel-level data could be encrypted directly to the auditor to avoid dependency on historical viewing keys.

Update: Resolved in [pull request #703](#) at commit [0b17527](#).

Low Severity

L-01 Signature Validation Occurs After Expensive Compilation in `__execute__`

In `__execute__`, signature validation (`assert_valid_signature`) is performed only after invoking `compile_actions`, which executes the full client-action compilation pipeline (including elliptic curve operations and Poseidon hashing). The `__validate__` function does not perform signature checks and only enforces OS-origin and zero-fee constraints (`tip == 0`, `max_price_per_unit == 0`).

This ordering allows unauthenticated inputs to reach the costly compilation stage before failing authentication, resulting in unnecessary resource consumption. However, this does not introduce a direct on-chain security risk. Mainnet sequencers are not affected due to the enforced zero-fee model, and any residual impact is limited to off-chain proving or simulation infrastructure.

Consider moving `assert_valid_signature` immediately after input parsing and before `compile_actions` in `__execute__` to ensure early rejection of invalid inputs and avoid unnecessary computation.

Update: Acknowledged, not resolved.

L-02 Nominal Token Accounting Breaks on Fee-On-Transfer and Rebasing Tokens

The protocol accounts for token movements using the nominal `amount` without verifying the actual balance change after transfer. In `deposit()`, the nominal amount is added to the `TokenBalances` ledger, and `_apply_transfer_from()` executes the ERC20 transfer without validating the received amount. In `_deposit_to_open_note()`, note state is written before the transfer executes using the nominal amount directly. The conservation check in `TokenBalancesImpl::assert_valid()` operates only on nominal values and does not reference actual token balances. Token addresses are not restricted to value-preserving ERC20 implementations, so non-standard token behaviours are not validated.

For fee-on-transfer tokens, the contract may receive less than the credited amount, leading to undercollateralization. For rebasing tokens, balances can change independently of protocol actions, causing recorded note values to diverge from actual contract balances over time. This breaks the core backing invariant and may result in failed withdrawals for honest users whose notes are backed by a shortfall created by earlier deposits. Rebasing tokens introduce additional risk, as these balance changes occur without transfers and cannot be detected via balance-delta checks.

Consider enforcing value-preserving behaviour by verifying that the actual balance increase equals the nominal amount and reverting otherwise. Since rebasing behaviour cannot be reliably detected on-chain, consider explicitly restricting supported tokens (e.g., via allowlisting) to standard ERC20 implementations.

Update: *Acknowledged, not resolved.*

L-03 Invalid Public Keys Trigger Generic Panic Instead of Explicit Error

`EcPointTrait::new_from_x(x: public_key)` is followed by `.unwrap()`, which produces a generic panic if the value is not a valid curve point. Other failure paths in the same function use named errors via `.expect(...)`, creating inconsistent error handling.

Failures become harder to diagnose and are inconsistent with the rest of the codebase. If an invalid public key ever enters storage, the resulting error will be opaque rather than actionable.

Consider replacing `unwrap()` with `.expect(errors::INVALID_PUBLIC_KEY)` and add a corresponding error constant to standardize error handling and improve diagnosability.

Update: Resolved in [pull request #702](#) at commit [b70e518](#).

L-04 `pack` Does Not Enforce 120-bit Constraint on `value_1`

The `pack` function combines two `u128` values into a `felt252` but does not enforce the implicit constraint that `value_1 < 2120`. This invariant is only checked downstream in `unpack`, creating a split that makes `pack` unsafe to reuse — future callers would not naturally be aware of the 120-bit requirement.

If `pack` is called with `value_1` in the range `[2120, 2123.8)`, the `try_into()` conversion succeeds silently. The violation only surfaces later when `unpack` is called, where it panics with `UNPACK1_OUT_OF_BOUNDS`; an error with no connection to the original call site. For values above `2123.8`, `PACK_OVERFLOW` fires, but this is an internal error constant that is opaque to both users and developers.

Consider enforcing `value_1 < 2120` directly within `pack` to validate inputs at the encoding boundary and provide clearer failure semantics. With this check in place, the upstream validation in `CreateEncNoteInput::assert_valid (salt < TWO_POW_120)` becomes a defense-in-depth layer rather than the primary guard, which aligns with proper validation layering. `PACK_OVERFLOW` can then be documented as a defensive safeguard or removed.

Update: Acknowledged, will resolve.

L-05 Ephemeral Secret Reuse in ECDH Encryption Breaks Channel Privacy

The ECDH encryption in `encrypt_channel_info` derives masks solely from `shared_x = f(r, recipient_public_key)`, where the ephemeral secret `r` is supplied by the client and only validated as non-zero. The contract does not enforce uniqueness or prevent reuse of `r`. If the same `r` is reused for the same recipient, the resulting masks are identical, leading to deterministic ciphertext reuse.

If `r` is reused, channel privacy collapses: channel keys and sender addresses can be recovered and note data becomes decryptable. The failure is silent and retroactive. The current SDK generates fresh randomness per action, so this is a defence-in-depth gap rather than an actively exploitable issue.

Consider mixing the existing `salt` field into the ECDH derivation to bind the shared secret per invocation. This ensures distinct masks even if `r` is reused and aligns with the pattern used elsewhere in the codebase.

Update: *Acknowledged, not resolved.*

Notes & Additional Information

N-01 Salt Type Inconsistency Across Encryption Hashes

The Encryption hash functions use inconsistent salt types: `compute_enc_token_hash` accepts `felt252`, while `compute_enc_amount_hash` uses `u128`. Both are cast to `felt252` before hashing, but the inconsistency is undocumented.

This can potentially introduce integration risk for off-chain components, as developers may assume a uniform salt type and produce mismatched hashes, leading to incorrect encryption/decryption or failure to locate notes.

Consider standardising the salt type across all encryption hash functions. If intentional, document the rationale to ensure correct off-chain implementations.

Update: *Resolved.*

N-02 Inconsistent Use of Reserved Slots in Hash Constructions

Some hash constructions include a reserved zero slot for forward compatibility, for example `compute_note_id`, while others at a similar abstraction level (e.g., `compute_channel_key`, `compute_channel_marker`, `compute_subchannel_marker`) omit it. No rationale is documented for this distinction.

This creates ambiguity about which hash constructions are intended to be extensible, increasing the risk of inconsistent assumptions in future development.

Consider standardising the pattern across all hash constructions, or explicitly documenting the rationale (e.g., extensible vs terminal hashes) to ensure consistent usage.

Update: *Acknowledged, not resolved.*

N-03 Comment Inconsistency on Viewing Key Mutability

Inline comments for `ClientAction::SetViewingKey` suggest that a viewing key can be replaced after registration. However, both the protocol specification (paper) and the implementation enforce immutability via `WriteOnce` storage.

This is a minor documentation inconsistency and may cause confusion.

Consider updating inline comments to reflect that viewing keys are immutable once registered.

Update: *Resolved in [pull request #704](#) at commit [e205062](#).*

N-04 Access Control Roles Marked as TODO in Sensitive Functions

Several sensitive functions use role checks that are marked with, indicating that the assigned roles are placeholders and intended to be updated before deployment. These include functions controlling the auditor key, fee parameters, and proof configuration. Several sensitive functions use role checks marked with [TODO comments](#). These include functions controlling the auditor key, fee parameters, and proof validity configuration. The comments indicate the assigned roles are placeholders intended to be updated before deployment.

This is not a correctness issue today, but unresolved role assignments may lead to unintended privileges if not finalized before deployment.

Consider finalizing all role assignments and removing TODO comments before deployment. Ensure the intended role model is clearly defined and consistently applied across all sensitive functions.

Update: *Resolved.*

Conclusion

The code under review implements a privacy-preserving token transfer protocol on Starknet, combining a UTXO-based design with off-chain proving and encrypted state management.

Overall, the engagement identified no critical- or high-severity issues. The findings primarily relate to trust assumptions, edge-case behaviors, and defense-in-depth improvements. Several recommendations were made to improve clarity around the security model, particularly with respect to centralized components such as the proving service, auditor role, and governance controls.

The Starkware team is encouraged to monitor off-chain components and governance operations, as these directly impact system integrity and user privacy.

The Starkware team was responsive throughout the engagement and provided clear documentation and context, which facilitated the review.

Appendix

Issue Classification

OpenZeppelin classifies smart contract vulnerabilities on a 5-level scale:

- Critical
- High
- Medium
- Low
- Note/Information

Critical Severity

This classification is applied when the issue's impact is catastrophic, threatening extensive damage to the client's reputation and/or causing severe financial loss to the client or users. The likelihood of exploitation can be high, warranting a swift response. Critical issues typically involve significant risks such as the permanent loss or locking of a large volume of users' sensitive assets or the failure of core system functionalities without viable mitigations. These issues demand immediate attention due to their potential to compromise system integrity or user trust significantly.

High Severity

These issues are characterized by the potential to substantially impact the client's reputation and/or result in considerable financial losses. The likelihood of exploitation is significant, warranting a swift response. Such issues might include temporary loss or locking of a significant number of users' sensitive assets or disruptions to critical system functionalities, albeit with potential, yet limited, mitigations available. The emphasis is on the significant but not always catastrophic effects on system operation or asset security, necessitating prompt and effective remediation.

Medium Severity

Issues classified as being of medium severity can lead to a noticeable negative impact on the client's reputation and/or moderate financial losses. Such issues, if left unattended, have a moderate likelihood of being exploited or may cause unwanted side effects in the system.

These issues are typically confined to a smaller subset of users' sensitive assets or might involve deviations from the specified system design that, while not directly financial in nature, compromise system integrity or user experience. The focus here is on issues that pose a real but contained risk, warranting timely attention to prevent escalation.

Low Severity

Low-severity issues are those that have a low impact on the client's operations and/or reputation. These issues may represent minor risks or inefficiencies to the client's specific business model. They are identified as areas for improvement that, while not urgent, could enhance the security and quality of the codebase if addressed.

Notes & Additional Information Severity

This category is reserved for issues that, despite having a minimal impact, are still important to resolve. Addressing these issues contributes to the overall security posture and code quality improvement but does not require immediate action. It reflects a commitment to maintaining high standards and continuous improvement, even in areas that do not pose immediate risks.