Code Review and Security Assessment For Light Protocol

April 25, 2022

Prepared For
Jorrit Palfner | GLAMOURPROPHECY LDA
Swen Schäferjohann | GLAMOURPROPHECY LDA

Prepared by
Er-Cheng Tang | HashCloak Inc
Thomas Steeger | HashCloak Inc
Mikerah Quintyne-Collins | HashCloak Inc

Table Of Contents

Executive Summary

GlamourProphecy LDA engaged HashCloak Inc for an audit of the LightProtocol which is a privacy protocol allowing shielded transactions on Solana. The audit was done with 3 auditors over 4 weeks, from February 14, 2022 to March 14, 2022. The relevant code base was the light-protocol-onchain repository, assessed at commit 26f060…4364fb. After our initial audit, the development team made additional changes to the codebase assessed at commit afcc90...85b345 on Apr 20, 2022. These changes were covered as well. The scope of the audit were all files in `light-protocol-onchain/program/src/`. During the first week we familiarized ourselves with the codebase and the cryptographic techniques used in the protocol. In the following weeks we investigated the security of the codebase through various efforts.

We found a variety of issues ranging from Critical to Low, and we provide some general guidance to improve the code quality.

Severity	**Number of Findings**
Critical	1
High	0
Medium	0
Low	2
Informational	1

Overview

LightProtocol is a privacy protocol on the Solana blockchain that allows users to transfer tokens anonymously. The main algorithmic ingredients behind the design of LightProtocol are the Groth16 zk-SNARK verifier and the Poseidon Merkle tree, which ensure both correctness and privacy of token transfers. To aid user experience, LightProtocol designs user accounts for users to store encrypted UTXOs onchain, giving them an overview of their private holdings.

The mechanism of anonymous token transfer goes as follows. On the one hand, users can mint tokens into the pool by generating secret notes and publishing their public commitments to the program. Commitments of all secret notes are compressed into a Merkle tree root that creates a sense of mixing. On the other hand, users can burn tokens from the pool by generating and submitting the nullifiers of the corresponding secret notes. The role of the nullifier is to ensure that users can withdraw their tokens only once. In every transaction, users have to specify the net flow between public and private tokens, and include a zero knowledge proof that validates the correctness of data. Users are responsible for balancing off the net flow amount by sending or receiving tokens to or from the pool. The zero knowledge proof has the effect of hiding all other information about the tokens being minted or burnt. As a result, there will be public token transfers into and out of the pool, but the connection between these separate transfers are hidden.

One of the main technical issues that LightProtocol solves is to implement the huge cryptographic calculations on the resource-restricted Solana runtime environment. To do so, they split the execution logic into smaller instructions and put efforts into defending possible attacks that may follow along.

The codebase has a hierarchical structure and can be divided into several logical units. On the root level, incoming instructions are mapped to their corresponding code paths depending on the instruction's byte code. The main module deals with amount validation and token transfer. The submodules deal with

- zero-knowledge proof verification
- hashing and merkle tree state management
- user account management

Methodology

We checked through common Solana vulnerabilities manually according to Neodyme's suggestions and Soteria's suggestions, and we ran the Soteria analyzer to aid our analysis. We further investigated potential attack surfaces based on our hacking experience. We checked if the cryptographic primitives were implemented correctly in comparison with known implementations (Poseidon hash and Groth16 zk-SNARK).

Claim Validity

Our investigation shows that the following claims made by LightProtocol are valid:

- Input accounts are properly checked throughout
- Temporary storage account cannot be manipulated
- Groth16 verifier is consistent with Arkworks' implementations (1, 2)
- Poseidon merkle root is computed with a correct instruction sequence
- Token transfer is implemented with correct amounts and accounts
- Double spending of the same note is prevented
- Anonymity is guaranteed for transactions that interact properly with shielded pool
- User account is implemented correctly

Findings

Modification of shared state is not an atomic operation

Type: Critical
Files affected: src/poseidon_merkle_tree (processor.rs, instructions.rs)

Description: The merkle tree account maintains a state shared by all users. Modification of such data should be done atomically to ensure data integrity. Atomic execution is usually achieved with locks, which prevents concurrent execution in the first place. The merkle tree structure has a lock that binds to the signer's public key, but such locking mechanism is incomplete. The lock should be binded to the temporary execution storage instead. In the current implementation, one can execute merkle tree instructions with distinct temporary storage accounts at the same time, as long as the accounts are initiated by the same signer. Therefore, the lock does not prevent concurrent merkle tree updates in certain cases. Moreover, there is state modification in the middle of the merkle tree instructions sequence (line `65` of instructions.rs), so the state modification is already non-atomic. Attacks that execute non-atomic operations in a concurrent way can cause a breakdown of state integrity and harm the security of the protocol.

Based on this issue, we propose the following attack that allows an attacker to insert a leaf into the state without ever paying for its face amount. First, the attacker sends `2` shielded transactions in sequence using the same account. For an honest transaction, there should be a total of `1502` instructions. However, the attacker will send both transactions with only the first `1501` instructions. Due to the inappropriate locking mechanism, the second transaction will encounter no errors, and the `filled_subtrees[]` field of the merkle tree will contain results calculated from the second transaction. Next, the attacker sends the last instruction of the first transaction. Because of the locking issue again, the instruction will be successful. The `next_index` field of the merkle tree is increased, but the `filled_subtrees[]` field remains. Till now, the attacker only pays for the first transaction, but the second transaction data is also incorporated into the merle tree state. Finally, the attacker sends a third shielded transaction in the whole. The new merkle tree root will be calculated using the saved `filled_subtrees[]`, which actually validates the UTXO of the second transaction. Therefore, the UTXO of the second transaction can be spent, but it was never paid for.

Impact: The above attack allows an attacker to steal an arbitrary amount from the shielded pool. We have implemented an attack demo in https://github.com/hashcloak/red-teaming_light-protocol.

Suggestion: In lines `159`, `170` of src/poseidon_merkle_tree/processor.rs, assign to the field `pubkey_locked` the key of `_tmp_storage_pda` instead of the key of `signer`. Modify the corresponding argument of `pubkey_check()` as well. This solves the concurrency issue. Also, we suggest moving the assignment in lines `65–66` of src/poseidon_merkle_tree/instructions.rs from the function `insert_1_inner_loop()` to the function `insert_last_double()`. This ensures atomicity of state change.

Status: Fixes are implemented at commits 870c07…8d4260 and b85065…9eee19. Now, the process of updating the merkle tree is considered secure.

Precomputed data is not validated

Type: Low
Files affected: src/utils/config.rs

Description: The array `ZERO_BYTES_MERKLE_TREE_18` contains magic bytes. They are supposed to represent precomputed poseidon hashes of the empty merkle tree, but it is unclear to code reviewers whether this is the case. Since the data is used as intermediate hashes for computing merkle roots, untrusty data may serve as potential trapdoors that allow improper withdrawal. We recommend adding tests to assure the readers that the array is indeed correctly generated.

Impact: It is not clear from the code whether the precomputed data are correct.

Suggestion: Add tests to validate the correctness of `ZERO_BYTES_MERKLE_TREE_18` and other magic numbers.

Status: A test that validates these magic numbers is implemented at commit 083c7a...577c52.

Inconsistent definition of the same data structure

Type: Low
Files affected: src/state.rs, src/groth16_verifier/prepare_inputs/state.rs

Description: The structure of the temporary storage account data is defined in multiple files with different packing and unpacking rules. Such coding style complicates data structure maintenance and is more likely to introduce errors. For example, the states `ChecksAndTransferState` and `PrepareInputsState` have inconsistent serialization of the fields `account_type`, `found_root`, `ext_amount`, `amount` and `nullifier_hash`. Misinterpretation of data fields can be dangerous. Fortunately, these errors do not lead to actual weakness of the current codebase.

Impact: The inconsistency can potentially lead to real problems in future development.

Suggestion: We recommend putting the complete de-/serialization logic into the `Pack` implementation of `TmpStoragePda`, without using `_unused` fields. This way the conversion between instruction bytecode and `TmpStoragePda` is located in a single trait implementation which increases the cohesion of the codebase and reduces possible future sources of errors. De-/serialization logic of other structs such as `ChecksAndTransferState` or `PrepareInputsState` can then be replaced by implementing `From/TryFrom` traits from Rust's standard library, converting the deserialized `TmpStoragePda` into said structs. In order to convert these structs back into a `TmpStoragePda` struct we recommend introducing a trait, which could look like:

```
trait IntoTmpStoragePda {
    fn into_tmp_storage_pda(&self, previous_pda: &TmpStoragePda) ->
TmpStoragePda;
}
```

Implementers could then use this trait, which allows to use the missing set of struct fields from the original `TmpStoragePda` to effectively create a new one consisting of the updated struct fields of the implementer and the missing ones from the original `TmpStoragePda`.

Status: The development team decides not to fix the issue at this moment.

Unused code and redundant instructions

Type: Informational
Files affected: src/groth16_verifier/final_exponentiation/instructions.rs, src/lib.rs

Description: Some instructions are defined but unused, while others are undefined but used. More specifically, the instruction set of final exponentiation contains instruction IDs `121`, `122`, which are unused and incorrect (changed variables in line `443` should be `F_F2_RANGE_ITER` instead). In addition, the instruction sequence `IX_ORDER` includes instruction ID `1` at the beginning of miller loop, but it is not defined in the instruction set of miller loop and is therefore skipped during execution.

Suggestion: We recommend removing unused codes and redundant instructions to make the codebase concise. If the redundant instruction is removed from `IX_ORDER`, we note that several constant indices also need to be adjusted.

Status: The development team decides not to fix the issue at this moment.

Recommendations

General coding style

When developing programs in Solana, the programmer has to map different instruction bytecodes to different program paths. This manual mapping of raw bytes is error-prone and one should try to mitigate this potential source of errors by using a concise coding style. For future iterations we recommend:

- Usage of Solana typical programming patterns or frameworks like e.g. Anchor.
- Improvement of code modularization increasing the code's cohesion and reducing its coupling and repetition.
- To encapsulate different branches of huge and nested if/else blocks into functions to improve the outline. Rust also offers a `match` statement, which is more suited in some situations.