OpenZeppelin | security

Zama Confidential Fungible Token Audit

May 22, 2025

Table of Contents

Summary

Type	
Timeline	From 2025-05-05 To 2025-05-16
Languages	Solidity

Total Issues	9 (7 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	0 (0 resolved)
Low Severity Issues	4 (3 resolved)
Notes & Additional Information	5 (4 resolved)

Scope

OpenZeppelin audited the OpenZeppelin/openzeppelin-confidential-contracts repository at commit e17ebd1.

In scope were the following files:

```
contracts
├── interfaces
│   ├── IConfidentialFungibleToken.sol
│   └── IConfidentialFungibleTokenReceiver.sol
├── token
│   ├── extensions/ConfidentialFungibleTokenERC20Wrapper.sol
│   ├── utils/ConfidentialFungibleTokenUtils.sol
│   └── ConfidentialFungibleToken.sol
└── utils
    └── TFHESafeMath.sol
```

Update: *All resolutions and the final state of the audited codebase mentioned in this report are contained at commit 71202e4.*

Please note that changes introduced to contracts not in-scope for this audit were not part of the additional reviews.

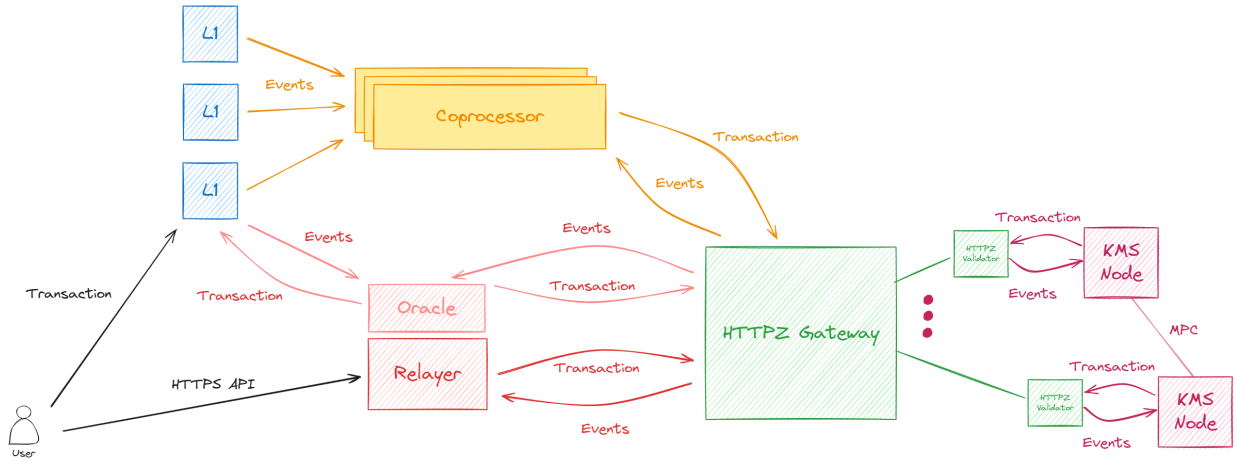
System Overview

Zama is a company that has built a Fully Homomorphic Encryption (FHE) coprocessor that works with an EVM-compatible chain. As such, it performs symbolic execution on-chain, while the real computation is done off-chain by the FHE coprocessors. Each encrypted variable on-chain is associated with a handle (or pointer), which is used to determine permissions for updating the value of the variable as well as decrypting and reencrypting.

From the Solidity smart contract side, Zama provides a library which contains functions that manipulate these handles and emit events such that the coprocessor will know which operations to perform for which handles. The Zama contracts also include an access control list (ACL) contract that determines the permissions for each handle, as well as a Gateway contract that facilitates decryption and reencryption.

Aside from the coprocessor and the smart contracts, there is a Gateway smart contract which runs on an Arbitrum L3 that mimics the permissions on the EVM-compatible chain. The validators for this L3 are Key Management System (KMS) nodes, which each contain a secret share of a private key. The public key is what all on-chain values are encrypted with, thus, the KMS nodes can jointly decrypt the values (though no individual node can decrypt by itself). Reencryption is similar to decryption, but adds an extra step to reencrypt that decrypted value with a different public key which only a specific user knows.

To decrypt an encrypted value on-chain, a function is called on the EVM-compatible chain, which emits an event. A relayer picks up this event and submits a decryption request to the Gateway on L3, which checks the permissions of the decryption request. If that address has the relevant permissions, the KMS nodes perform the Multi-Party Computation (MPC) decryption of that value and the KMS nodes also sign their decryption. The relayer then obtains this information from the L3 (which includes the signatures) and calls the EVM-compatible chain, where a smart contract validates these signatures. The architectural diagram below illustrates this process flow.



This specific audit was focused on two contracts, a confidential ERC-20 token in `ConfidentialFungibleToken.sol`, and a confidential wrapper for an existing non-confidential token in `ConfidentialFungibleTokenERC20Wrapper.sol`. Both token contracts leverage the Zama `TFHE` library for encryption to ensure transaction privacy, which allows for operations on encrypted values without revealing the actual amounts, ensuring the confidentiality of token amounts during transfers. The token contracts also contain an asynchronous operation for users to begin the disclosure process of their encrypted amounts, which is subsequently picked up off-chain, decrypted, and finally relayed back on-chain, finalizing the disclosure process.

Moreover, the contracts have ERC-1363-like functionality with a transfer-and-call pattern. The wrapper token contact extends `ConfidentialFungibleToken` by taking an underlying non-confidential ERC-20 token and wrapping the user-specified amount into a confidential token. It also implements the `IERC1363Receiver` interface to allow users to transfer ERC-1363 tokens directly to the wrapper with a callback to wrap the tokens.

Security Model and Trust Assumptions

The Zama architecture relies on certain trust assumptions. First, users must trust that the coprocessors are honestly performing their FHE operations on the encrypted data. There is no validity proof submitted by these coprocessors and, thus, users are trusting that they are not simply computing a set of arbitrary functions on the encrypted data, but rather the ones read from the event. Secondly, users must trust that the KMS nodes are not colluding. Since each

KMS node has a share of the private key, if enough of these nodes collude, they can decrypt the encrypted data for themselves, which would lead to them learning all the secret values.

There is also a trust assumption that the coprocessors update the Gateway L3 with all permanent permissions on L1. Without this, users would not be able to successfully decrypt or reencrypt their values when needed. Similarly, it is trusted that the coprocessors correctly verify the ZKP of encryption for all user inputs, such that they only allow correctly encrypted inputs to be submitted on-chain.

A more subtle trust assumption is that the relayer submits the decryption requests to the L3 and also returns the decrypted data back to the EVM-compatible chain. While the relayer cannot submit false data as the KMS node signatures are checked on-chain, refusal to submit any requests could lead to a loss of liveness. However, this can be prevented by users running their own relayer.

Privileged Roles

The `Gateway` contract has a privileged role as it is the only address that can finalize the disclosure of an encrypted amount for a confidential token. Similarly, the `Gateway` contract is the only entity that can finalize the unwrapping of a wrapped confidential token, which includes disclosing the encrypted amount and transferring that back to the unwrapped version of the ERC-20.

Low Severity

L-01 Misleading Pragma

The `ConfidentialFungibleToken` and the `ConfidentialFungibleTokenERC20Wrapper` contracts have the following pragma: `pragma solidity ^0.8.24;`. However, the code uses `require` with custom errors in multiple locations, for example here and here, which was introduced in Solidity 0.8.26. Therefore, this contract cannot be compiled with a version below `0.8.26`.

Consider correcting the pragma to start from version `0.8.26`.

Update: *Resolved in pull request #33 at commit 38bd511.*

L-02 Misleading Documentation

Throughout the codebase, multiple instances of misleading comments were identified:

- The comment in lines 65-66 of `IConfidentialFungibleToken.sol` states that the "caller must already be approved by ACL", but this should be changed to "allowed" as the ACL does not do approvals.
- The comment in lines 84-85 of `IConfidentialFungibleToken.sol` states that the "caller must be already be approved by ACL", but this should be changed to "allowed" as the ACL does not do approvals.

Consider revising the aforementioned comments to improve consistency and more accurately reflect the implemented logic, making it easier for auditors and other parties examining the code to understand what each section of code is designed to do.

Update: *Resolved in pull request #34 at commit 57f40ac.*

L-03 Deviation from ERC-1363 Specification

The implementation of the `_transferAndCall` function deviates from the expected behavior outlined in the ERC-1363 specification, particularly in its interaction with EOAs. According to the current implementation, the `_transferAndCall` function invokes

`checkOnERC1363TransferReceived`, which subsequently calls `onConfidentialTransferReceived`. When this call is made to an EOA (which has not delegated to a smart contract using ERC-7702), the operation results in a no-op (no operation), effectively allowing the transaction to proceed without reverting.

This behavior contrasts with the guidance provided by the author of the ERC-1363 specification in a discussion on the EIP-1363 pull request, where it is explicitly stated that `transferAndCall` "MUST revert if the recipient is an EOA address, ...". This clarification implies that the intended behavior for transactions targeting EOAs is to fail, ensuring that tokens are not transferred without the recipient contract's acknowledgment of the transfer.

To align with the ERC-1363 specification and the clarifications provided by its author, consider modifying the implementation of `onConfidentialTransferReceived` to ensure that it reverts when attempting to transfer tokens to an EOA, ensuring that it only succeeds when interacting with contracts prepared to handle them, according to the ERC-1363 protocol.

Update: *Acknowledged, not resolved. The Solidity Contracts team stated:*

> *We prefer to have transfers with callback to EOAs be no-ops, diverging from the ERC-1363 standard. We will remove mentions to the standard given non-compliance.*

L-04 Enhancement of the `_tryGetAssetDecimals` Function for Stricter Length Validation

The `_tryGetAssetDecimals` function is designed to retrieve the decimals value from ERC-20 tokens by making a `staticcall` to the token's decimals function. The current implementation checks that the call was successful and that the length of the `encodedDecimals` result is at least 32 bytes. This approach, while ensuring that the returned data is not shorter than expected, does not account for scenarios where additional, potentially irrelevant data is appended to a valid ABI-encoded `uint8` decimals value, thereby leaving room for non-standard responses that still pass the check.

Consider adjusting the length check to enforce an exact length of 32 bytes for the `encodedDecimals` result to enhance the robustness and predictability of the `_tryGetAssetDecimals` function.

Update: *Resolved in pull request #41 at commit 75cdc88.*

Notes & Additional Information

N-01 Superfluous Allowance

In the `confidentialTransfer` function of `ConfidentialFungibleToken.sol`, the `allowTransient` function is called on the `transferred` variable for `msg.sender`. However, in the previous line of code that calls `_transfer`, the `msg.sender` is already given a permanent allowance since the `msg.sender` is the `from` address in this case. Thus, the `allowTransient` function call is unnecessary.

Furthermore, in the `_transferAndCall` function, the `allowTransient` function is called on the `sent` variable for `to`. However, in the previous line of code that calls `_transfer`, `to` is already given a permanent allowance as the `_transfer` function ensures that the `to` address is not the zero address.

Consider removing both of the `allowTransient` calls as they are superfluous.

Update: *Resolved in pull request #32 at commit 96e07c2.*

N-02 Lack of Getters for Decryption Requests

In the implementations of `ConfidentialFungibleToken.sol` and `ConfidentialFungibleTokenERC20Wrapper.sol`, there is a noticeable inconsistency in the accessibility of certain storage variables. Specifically, while getters have been implemented for most `private` storage variables, the `_requestHandles` variable in `ConfidentialFungibleToken.sol` and the `_receivers` variable in `ConfidentialFungibleTokenERC20Wrapper.sol` lack corresponding `public` getter functions. This omission restricts external visibility into these variables, potentially limiting the ability of external contracts or users to interact with or fully understand the state of these contracts.

Consider implementing a `public` getter function for the `_requestHandles` and `_receivers` variables. This would enhance transparency and allow external contracts and interfaces to retrieve and display the current list of requests.

Update: *Acknowledged, not resolved. The Solidity Contracts team stated:*

> *The PR won't be merged at the moment. We are unable to identify a valid usecase where the getter would be utilized externally or overridden.*

N-03 Inconsistent Order Within a Contract

The `ConfidentialFungibleTokenERC20Wrapper` contract deviates from the Solidity Style Guide due to having an inconsistent ordering of functions. Concretely, it has a `private` function before the `public` ones.

To improve the project's overall legibility, consider standardizing ordering throughout the codebase as recommended by the Solidity Style Guide (Order of functions).

Update: *Resolved in pull request #35 at commit 5bc44fc.*

N-04 Inconsistent Library Syntax Usage

There is an observable inconsistency in the syntax used for invoking library functions. Specifically, the manner in which library functions are called varies between different files, leading to a lack of uniformity across the codebase. For instance, in `ConfidentialFungibleTokenUtils.sol`, library functions are called using the syntax `x.asEbool()`. Conversely, in `TFHESafeMath.sol`, the syntax used is `TFHE.asEbool()`. In addition, in the Zama example ERC-20, the syntax follows the use of prefixing the library name. This discrepancy in syntax style not only hampers readability but also increases the cognitive load on developers and auditors trying to understand or review the code.

Consider standardizing the syntax for library function calls across all files using the prefixed library name (`TFHE.asEbool()`). Applying it consistently will improve the overall quality and coherence of the code.

Update: *Resolved in pull request #42 at commit 4015ae7.*

N-05 Multiple Optimizable State Reads

In `ConfidentialFungibleToken.sol` multiple opportunities for optimizing storage reads were identified:

- In `finalizeDiscloseEncryptedAmount`, the `_requestHandles` storage variable is read three times.
- In `_update`, the `_balances[from]` storage variable is read twice.

Consider reducing SLOAD operations that consume unnecessary amounts of gas by caching the values in a `memory` variable.

Update: *Resolved in pull request #36 at commit 51a737a.*

Conclusion

The Zama Confidential Fungible Token contracts introduced the ability to have a confidential token as well as confidentially wrap a non-confidential ERC-20. The Solidity Contracts team and the Zama team are commended for addressing user needs by incorporating confidentiality into tokens.

During the audit, particular care was taken to document edge cases, ensuring that integrators are informed of potential risks when interacting with these contracts. Furthermore, permissions were granted in a granular yet strict way, allowing only the necessary parties to be able to decrypt and reencrypt data. Such efforts aim to create a more resilient codebase, recognizing the role of these contracts as a foundational component within the blockchain ecosystem. The Solidity Contracts team and the Zama team have demonstrated a strong commitment to maximizing the security of these contracts, and the audit team is glad to have collaborated with them on this milestone.