OpenZeppelin | security

Zama Confidential Vesting Wallet and Voting Audit

August 12, 2025

Table of Contents

Summary

Type	Library
Timeline	From 2025-07-14 To 2025-07-25
Languages	Solidity

Total Issues	21 (15 resolved, 1 partially resolved)
Critical Severity Issues	1 (1 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	4 (4 resolved)
Low Severity Issues	10 (6 resolved)
Notes & Additional Information	5 (3 resolved, 1 partially resolved)

Scope

OpenZeppelin audited the OpenZeppelin/openzeppelin-confidential-contracts repository at commit 432762c. This commit was diffed against commit 71202e4 and all new files were fully audited. The contents of the `temporary-Checkpoints.sol` file had been copied over from the `Checkpoints.sol` file in OpenZeppelin/openzeppelin-contracts repository, and the diff for this file at commit b84db20 was audited.

In scope were the following files:

```
contracts
├── finance
│   ├── ERC7821WithExecutor.sol
│   ├── VestingWalletCliffConfidential.sol
│   ├── VestingWalletConfidential.sol
│   └── VestingWalletConfidentialFactory.sol
├── governance
│   └── utils/VotesConfidential.sol
├── interfaces
│   ├── IConfidentialFungibleToken.sol
│   └── IConfidentialFungibleTokenReceiver.sol
├── token
│   ├── extensions
│   │   ├── ConfidentialFungibleTokenERC20Wrapper.sol
│   │   └── ConfidentialFungibleTokenVotes.sol
│   ├── utils
│   │   └── ConfidentialFungibleTokenUtils.sol
│   └── ConfidentialFungibleToken.sol
└── utils
    ├── structs
    │   ├── CheckpointsConfidential.sol
    │   └── temporary-Checkpoints.sol
    └── TFHESafeMath.sol
```

The final state of the audited codebase, including all implemented resolutions, is reflected in commit edda7dd.

System Overview

Zama's architecture integrates a Fully Homomorphic Encryption (FHE) coprocessor with an EVM-compatible blockchain to bring confidentiality to smart contracts. This audit focuses on updates to the confidential fungible token and a new suite of confidential contracts for vesting and voting, all built using Zama's FHE technology. Key changes to the confidential fungible token include a reduction in decimal precision from 9 to 6 and the integration of the latest Zama library and system.

The vesting contracts enable the creation of confidential vesting schedules, where the token amount is encrypted and released to the beneficiary over time. The base contract supports linear vesting by default, while the `VestingWalletCliffConfidential` variant introduces a cliff-based release mechanism. These contracts have been designed to be deployed as clones via the `VestingWalletConfidentialFactory` contract. Although optional, they are intended to support the staking functionality. To facilitate this, the `ERC7821WithExecutor` contract permits a privileged entity to transfer tokens from the vesting wallet to a staking contract.

The audit also covers confidential voting contracts. The `VotesConfidential` contract tracks encrypted vote balances, including delegated votes. It is designed to work alongside a token contract, such as `ConfidentialFungibleTokenVotes`, which inherits from the `ConfidentialFungibleToken` contract. Supporting libraries and interfaces were also reviewed as part of this audit.

Security Model and Trust Assumptions

As noted in previous audits, the Zama architecture depends on trust assumptions regarding the Zama coprocessors and the KMS nodes. Specifically, it is assumed that the coprocessors have correctly updated the Gateway (on the Arbitrum chain) with all permanent permissions on L1. Additionally, there is an implicit assumption that the coprocessors correctly verify the zero-

knowledge proofs (ZKPs) of encryption for all user inputs, ensuring that only properly encrypted inputs are allowed to be submitted on-chain.

Regarding the vesting contracts, it is assumed that the executor is a trusted party that can move tokens from the vesting wallet to a staking contract. The executor is trusted to only perform staking and unstaking and not to make arbitrary transfers to bypass the vesting schedule.

Assumptions When Using the Library

The confidential contracts under review have been designed as a library for other projects to build upon. As a result, it is important for any project looking to use these contracts to be able to use them in the manner that they have been designed, paying particular attention to Zama's FHE coprocessing architecture. As mentioned previously, the coprocessor architecture involves symbolic execution, meaning that the FHE operations are not conducted synchronously in a transaction, but later on off-chain. Consequently, oftentimes the design patterns for a failure do not include a `revert` during the transaction (as the actual computation is performed later off-chain), but rather an `FHE.select` that will perform a no-op.

This must be taken into careful consideration, as "failed transactions" still often result in a completed transaction. Special care must be taken when calling multiple functions or performing multiple operations in the same transaction. For example, it is possible that the first operation succeeds, while the second one fails, and it is important to rollback the first operation (or any previous operations if more than two) in order to maintain a consistent state.

Furthermore, it is important to consider the role of allowance when using these contracts. Any contracts that require permission to perform FHE operations on any handles owned by other contracts must be granted permission. For example, the `VotesConfidential` contract keeps track of the voting power for different addresses and has an allowance on all of these handles. However, a governance contract that would need to access the voting power stored in this `VotesConfidential` contract would need allowance for these handles as well to be able to sum up the total amount of votes for a certain proposal. If this is the desired behavior, this granting of allowance must be explicitly handled in the child contract that is inheriting from `VotesConfidential`.

Critical Severity

C-01 Invalid Return Value on Transfers With Callback

The `confidentialTransferAndCall` and `confidentialTransferFromAndCall` functions use the `internal` `_transferAndCall` function. This function incorrectly returns zero on successful non-zero transfers if the recipient transfers tokens out during the on-received call. This results in the external-facing functions also incorrectly returning zero.

This happens because `_transferAndCall` first optimistically transfers the tokens before invoking `checkOnTransferReceived`, which in turn calls the `onConfidentialTransferReceived` function of the recipient. The recipient then returns a boolean value of their choosing. If the returned boolean is `false`, the computed transferred amount is zero, and the initial transferred amount is refunded.

The issue arises when the recipient transfers the tokens out in the middle of the `onConfidentialTransferReceived` call to another address (possibly one that they control) and returns `false`. As such, the computed transferred amount will be zero, and the refund will be attempted to be processed. However, as the recipient's balance is now less than the initial transferred amount since they have already transferred out their tokens, the refund will silently fail (in other words, will refund 0 tokens).

The code, as it is currently written, assumes that the refund can never fail. Hence, it does not check if the refund was successful. Since the tokens are not refunded to the sender, the recipient is able to keep the tokens (in the other address), while returning zero as the transferred amount, which is then propagated to the calling functions. This behavior can be abused if any contract relies on the return value of the transfer functions.

Consider checking the return value from the refund and setting the `transferred` amount to the original `sent` value if there was nothing refunded.

Update: *Resolved in pull request #118 and applied at commit 2100c93.*

High Severity

H-01 Wallet Can Vest Ahead of Schedule Due to Overflow

This `VestingWalletConfidential` contract tracks the cumulative released amount of each token in a storage variable of type `euint64`. When the released tokens are sent back into the vesting wallet (for example, by the owner) and released again, the contract blindly increments the released counter without detecting overflow.

As `euint64` can wrap around its maximum value, an owner can cycle tokens back into the wallet and trigger multiple `release()` calls to overflow the released counter. Once the counter wraps, subsequent vesting calculations treat the released amount as a small number, allowing tokens to vest ahead of schedule.

Consider the following scenario:

1. At 75% elapsed time, the wallet holds its max `euint64` balance and has released zero.
2. Calling `release()` moves 75% of the tokens out (the wallet now has 25% of the tokens, with 75% still released).
3. Owner transfers 35% back in (the wallet now has 60% of the tokens, with 75% having been released).
4. Calling `release()` again computes (60% + 75%) × 75% - 75% = 26.25%, updating released to 101.25%, which overflows to 1.25%.
5. A third `release()` then vests (33.75% + 1.25%) × 75% - 1.25% = 25%, giving the owner a total of 91%—far above the intended 75%.

By wrapping the released counter, the owner gains an extra 16% of tokens ahead of schedule.

Consider storing and manipulating the released amount in a `euint128` container. This change ensures that the released counter remains monotonic and preserves the integrity of the vesting schedule.

Update: *Resolved in pull request #120 and applied at commit 54e322d and in pull request #149 and applied at commit edda7dd.*

Medium Severity

M-01 Inconsistent and Error-Prone Initialization Functions

The base wallet initializer performs chained initialization by invoking the initializer of the inherited `OwnableUpgradeable` contract. In contrast, the cliff wallet initializer performs unchained initialization and does not invoke the initializer of the inherited base wallet contract. Furthermore, both initializers (the one in the base wallet and the cliff wallet) use identical naming conventions (do not contain the word "unchained") even though they have different behaviors. This inconsistency creates both limitations and potential errors.

The lack of a chained initialization function in the cliff wallet contract is particularly problematic because the initialization behavior is order-dependent. As such, if a wallet implementation were to call the cliff wallet initializer before the base wallet initializer, it would result in a revert due to unset variables. While the mock wallet implementation calls the initializers in the correct order, other implementations could easily mix up the sequence.

Consider implementing both chained and unchained initializers in both abstract wallet contracts, with explicit names that differentiate between the two approaches. Also consider using and recommending the use of the chained initializer in contracts that inherit from the cliff wallet.

Update: *Resolved in pull request #119 and applied at commit 6531d44.*

M-02 Over-Minting When Wrapping Fee-on-Transfer Tokens

In the `wrap` function of `ConfidentialFungibleTokenERC20Wrapper.sol`, the contract transfers `amount - (amount % rate())` underlying tokens from the user to the wrapper contract. However, later on in the same function, tokens are minted as if the full amount had been received. While this logic may hold for standard tokens, in the case of fee-on-transfer tokens, the wrapper will receive fewer underlying tokens than it is minting. This would break the peg and render the wrapper insolvent if everyone tried to unwrap.

Consider adding documentation that fee-on-transfer or other deflationary-type tokens are not supported by this wrapper.

Update: *Resolved in pull request #134 and applied at commit d17caee.*

M-03 Underflow in Releasable Calculation When Combined with Executor-Based Staking

This `VestingWalletConfidential` contract is intended to operate alongside `ERC7821WithExecutor`, which grants an executor permission to move tokens out of the vesting wallet for activities such as staking. When the executor withdraws tokens that have not yet vested, the on-chain balance decreases while the vesting schedule's historical allocation remains unchanged.

The linear vesting schedule computes vested tokens by multiplying the total allocation by the fraction of elapsed time. If an executor withdrawal or negative rebase reduces the balance below the already released amount, the computed vested amount becomes smaller than the released amount, resulting in an underflow.

Underflow in this context always yields a result that exceeds the contract's actual balance. This means that the confidential token contract transfers zero, leaving all state unchanged. While this behavior prevents unintended token movements, relying on the token transfer failure undermines the semantic guarantee of a non-negative releasable amount and could mask genuine balance inconsistencies.

To enforce correct behavior at the point of calculation, consider performing the subtraction using the `tryDecrease` function from the `TFHESafeMath` library.

Update: *Resolved in pull request #122 and applied at commit 2606fc2.*

M-04 Delegatees Can Deduce the Balances of Their Delegators

The `VotesConfidential` contract records encrypted vote balances and checkpoints for each delegate. When delegation changes occur, whether via token transfers or explicit `delegate` calls, the contract updates both the old and new delegate checkpoints and automatically grants each party the permission to decrypt its own updated balance. Since a delegatee can decrypt its checkpoint after every move, it can observe exactly how much its own balance has changed. In the context of `ConfidentialFungibleTokenVotes`, this means that the delegatee would know all the balances and balance changes of its delegators.

Consider removing both `FHE.allow()` invocations in the `_moveDelegateVotes` function and deferring any decision to reveal individual or aggregate balances to the inheriting contract.

Update: *Resolved in pull request #143 and applied at commit a713b1f. The team stated:*

> *Default allowances have been removed and can be re-added as necessary by the developer via the `HandleAccessManager`.*

Low Severity

L-01 Flipped Event Parameter

The `VestingWalletConfidentialCreated` event takes in six parameters, starting with `vestingWalletConfidential` and `beneficiary`. However, in the `createVestingWalletConfidential` function, the event is emitted starting with `beneficiary` followed by the `vestingWalletConfidentialAddress`, which might mislead a service reading the events as the two fields are flipped.

Consider switching the order in the event emission to have the first input be `vestingWalletConfidentialAddress` and the second one `beneficiary`.

Update: *Resolved in pull request #121 and applied at commit 4ffb93a. The team stated:*

> *Note that the event was modified later--it now only has 2 parameters.*

L-02 User Unable to View Vesting Amount Without Release

The `VestingWalletConfidential` contract is meant to be deployed using a clone, which sends it tokens that are then slowly able to be released to a user according to a vesting schedule. Since the vesting wallet is the address that is funded with tokens using `confidentialTransferFrom`, it will have allow permissions for the tokens. However, since the user (owner) will be at a different address, they do not have allow permissions and will be unable to decrypt the total balance of the vesting wallet. This means that the user has no way of knowing if the vesting wallet was funded with the correct amount as intended.

Consider implementing a separate function in `VestingWalletConfidential` that performs an `FHE.allow` call to give the owner allow permissions for any arbitrary handle.

Update: *Resolved in pull request #143 and applied at commit a713b1f. The team stated:*

> *The `HandleAccessManager` can be used if this is a needed feature.*

L-03 Lack of Checks-Effects-Interactions (CEI) Ordering

The `onTransferReceived` function of `ConfidentialFungibleTokenERC20Wrapper.sol` first performs a check, followed by a transfer of the excess (if there is any), and then a mint. The external call in this function is `SafeERC20.safeTransfer`, which is a defined function in the OpenZeppelin Contracts library that involves calling the `transfer` function of the underlying token. This `transfer` function could be arbitrarily malicious and cause reentrancy. However, it is assumed that the wrapper contract trusts that the underlying contract is not malicious, as its sole purpose is to be used as a confidential version of the underlying token.

Consider adopting the CEI pattern by minting the tokens prior to transferring the excess back to the sender.

Update: *Resolved in pull request #125 and applied at commit ba873f0.*

L-04 Wallet Factory Lacks Support for Custom Wallet Implementations

The `VestingWalletConfidentialFactory` contract allows adding an arbitrary address for the wallet implementation. To support a variety of wallet implementations, the `_initializeVestingWallet` function is left to be implemented by the inheriting contract. However, this unimplemented function only accepts a fixed set of arguments used by the `VestingWalletCliffConfidential` initializer. This design does not accommodate potential wallet implementations that might require different arguments in their initializers.

Consider adding a `bytes` argument to the `_initializeVestingWallet` function, such that the overriding function can decode it to support any extra arguments for initialization. This change also requires including the `bytes` argument when generating the salt.

Update: *Resolved in pull request #135 and applied at commit 3988842 and in pull request #150 and applied at commit dce017a. The team stated:*

> *We opted to wrap all parameters into this single bytes arg.*

L-05 Lack of Overflow and Underflow Checks in `_transferVotingUnits`

The `_transferVotingUnits` function of the `VotesConfidential` contract is intended to be called by a child contract after an update. When the update is caused by a mint or a burn, the vote checkpoint is updated by calling the `confidentialTokenSupply` function, which assumes that the supply is updated prior to this call. Subsequently, for any mint, burn, or transfer, the `_moveDelegateVotes` function is called as well. While the warning in the comment is helpful, and the supply is updated properly in the `ConfidentialFungibleTokenVotes` contract, the warning still does not provide assurance that others using the library will perform the required updates prior to this function call, which could lead to over or underflows.

Consider using the `tryIncrease` or `tryDecrease` functions instead of `add` and `sub` to prevent overflow and underflow. Additionally, consider returning the actual transferred voting units to allow the inheriting contracts to use it.

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

> *We opt not to implement the suggested fix--when integrated correctly, the additional code is unnecessary and consumes additional gas. Note that even if the suggested fix is implemented, integrators who update their token state after moving delegations will still be breaking the total supply checkpoints.*

L-06 Different Pragma Directives Being Used

In order to clearly identify the Solidity version with which the contracts will be compiled, pragma directives should be fixed and consistent across file imports.

Throughout the codebase, multiple instances of different pragma directives being used were identified. Most notably, some directives that state `pragma solidity ^0.8.27;` while others state `pragma solidity ^0.8.24;`. In order to leverage the use of custom errors in `require` statements, which has been supported since Solidity version 0.8.26, consider updating all files to have the pragma version `^0.8.27;`.

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

> *Pragmas are intentionally chosen to indicate the earliest version of solc which will properly compile the given contract. One out of scope contract had the incorrect pragma according to this rule and was fixed.*

L-07 Missing and Misleading Documentation

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

- The comment above the `finalizeDiscloseEncryptedAmount` function in `ConfidentialFungibleToken.sol` states that `finalizeDiscloseEncryptedAmount` "may only be called by the gateway contract". However, this function can be called by any address.
- The comment above the `finalizeUnwrap` function in `ConfidentialFungibleTokenERC20Wrapper.sol` states that this function is "called by the fhEVM gateway". However, this is no longer necessarily true.
- The comment above the interface `IConfidentialFungibleToken` states that the token contract utilizes the Zama TFHE library. However, this library has been renamed to FHE.
- The file and corresponding library `TFHESafeMath.sol` could be renamed to `FHESafeMath.sol` to reflect the name change of the Zama library.
- The comment in line 86 states that the `vestedAmount` function "calculates the amount of tokens that have already vested". However, it should say "calculates the amount of tokens that have been vested at the given timestamp", as this function can be called at any timestamp that is provided as input.

Furthermore, multiple instances of missing docstrings were identified:

- In `VestingWalletConfidential.sol`, the `VestingWalletConfidentialTokenReleased` event does not have any accompanying docstrings.
- In `VestingWalletConfidential.sol`, the fact that this contract is meant to be deployed as a clone is not mentioned.
- In `VestingWalletConfidentialFactory.sol`, the `VestingWalletConfidentialFunded` event does not have any accompanying docstrings.
- In `VestingWalletConfidentialFactory.sol`, the `VestingWalletConfidentialCreated` event does not have any accompanying docstrings.
- In `VotesConfidential.sol`, the `VotesConfidential` abstract contract does not have any accompanying docstrings.
- In `ConfidentialFungibleTokenVotes.sol`, the `ConfidentialFungibleTokenVotes` abstract contract does not have any accompanying docstrings.

- In `ConfidentialFungibleTokenVotes.sol`, the `confidentialTotalSupply` function does not have any accompanying docstrings.

Consider adding or 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 #137 and applied at commit 5ef0b81.*

L-08 Ambiguous Call to Parent Contract

Within the `ConfidentialFungibleTokenVotes` contract, the `super.confidentialTotalSupply` function call is ambiguous. In this case, the `super` refers to the `ConfidentialFungibleToken` contract because the function is not implemented in `VotesConfidential`. However, if the function were implemented in `VotesConfidential` then `super` would refer to `VotesConfidential`, even without any other changes to the code.

Consider avoiding ambiguous calls to parent contracts. Instead, explicitly specifying which parent contract's function is being called.

Update: *Nullified as super is used on purpose. The team stated:*

> *We utilize super instead of an explicit parent call to ensure that we don't miss any logic in the function within the inheritance chain. If a function were added to `VotesConfidential` at a later point in time, we would want that function to be called.*

L-09 Funds Can Be Lost Due to Incorrect Input Parameters

The `batchFundVestingWalletConfidential` function of the `VestingWalletConfidentialFactory` contract predicts the wallet addresses based on `create2` address derivation. If any input parameter is slightly incorrect, the derived address will not point to the intended wallet address. In that case, the derived address is likely to be an empty address, in which case the transferred tokens would become locked.

To reduce the risk of transferring funds to incorrect addresses, consider checking the code size of the derived addresses prior to transferring tokens and reverting if they have no code.

Update: *Nullified as a use case for funding wallets prior to deployment exists. The team stated:*

> *The derived vesting wallet address can be deployed at any point using the initialization data. Users should be sending tokens to uninitialized accounts for their clients to claim them at a later point in time.*

L-10 Potential Lack of Uniqueness of Salt

In the `_getCreate2VestingWalletConfidentialSalt` function of the `VestingWalletConfidentialFactory` contract, the parameters for the salt are the `beneficiary`, `startTimestamp`, `durationSeconds`, `cliffSeconds`, and `executor`. These parameters help determine the vesting wallet address. In the event that someone deploying multiple vesting wallets would like to create two or more vesting wallets with the same parameters, they would not be able to.

Consider having an extra parameter (e.g., `uniqueId`) that can be set to an arbitrary value to serve as a nonce in the event that multiple vesting wallets with the same parameters have to be created.

Update: *Resolved in pull request #135 and applied at commit 3988842 and in pull request #150 and applied at commit dce017a. The team stated:*

> *Users can add entropy to the end of the initialization now.*

Notes & Additional Information

N-01 Assembly Blocks Can Be Marked as `memory-safe`

Several assembly blocks in the codebase only perform slot assignment operations that do not interact with the memory, making them candidates for the `memory-safe` annotation. This annotation allows the Solidity compiler to perform additional optimizations by indicating that the assembly block does not read from or write to memory in ways that could interfere with Solidity's memory management.

The following assembly blocks only assign storage slot values and can be safely marked as `memory-safe`:

- VestingWalletCliffConfidential.sol:58
- VestingWalletConfidential.sol:126
- ERC7821WithExecutor.sol:42

Consider adding the `memory-safe` annotation to the above-listed assembly blocks to enable compiler optimizations. In addition, consider clearly documenting their memory safety properties.

Update: *Resolved in pull request #132 and applied at commit 480b2af.*

N-02 Inconsistency in Data Types

In the `VestingWalletConfidential` contract, the `VestingWalletStorage` struct has `_start` and `_duration` fields that are `uint64`. In the initializer, the corresponding parameters are of type `uint48`.

While the implicit cast prevents any overflow issue, consider using a consistent type to prevent any subtle errors in the future. Furthermore, consider updating the factory contract to match the type, as the `startTimestamp` and `durationSeconds` are also of type `uint48`. Similarly, the `cliffSeconds` field is of type `uint48`, while in `VestingWalletCliffConfidential`, the `_cliff` field is of type `uint64`.

Update: *Partially resolved in pull request #146 and applied at commit 0d953c9. The team stated:*

> *We've swapped data types to ensure that all timestamp function parameters are `uint48`s. We left storage for these variables at uint64, given that there is 0 cost involved, and it ensures overflows are impossible.*

N-03 Unused Error

In `CheckpointsConfidential.sol`, the `CheckpointUnorderedInsertion` error is unused.

To improve the overall clarity, intentionality, and readability of the codebase, consider either using or removing any currently unused errors.

Update: *Resolved in pull request #131 at commit 14f3e43.*

N-04 Use Bit Shifting Instead of Multiplication

The `_unsafeAccess` function uses `mul(pos, 2)` instead of the more gas-efficient `shl(1, pos)` operation. Left-shifting by 1 bit is equivalent to multiplying by 2 but consumes less gas.

Consider replacing `mul(pos, 2)` with `shl(1, pos)` to reduce gas consumption.

Update: *Acknowledged, will resolve. The team stated:*

> *This file is copied exactly from the vanilla repo, so we will not change it here. We will consider this change in vanilla at a later point in time.*

N-05 `ConfidentialFungibleTokenERC20Wrapper` Assumes 18 Decimals for Tokens

The `ConfidentialFungibleTokenERC20Wrapper` contract assumes that the underlying token has 18 decimals when the token does not implement the optional `decimals` view function. This can lead to unexpected rates and to the fractionalizing of zero-decimal tokens.

Consider documenting this feature and additionally allowing the inheriting contract to explicitly set the underlying token decimals by updating the visibility of the `_tryGetAssetDecimals` function from `private` to `internal`.

Update: *Resolved in pull request #133 at commit 124894b.*

Conclusion

The reviewed changes update the Zama vesting and voting contracts to introduce the ability to have confidentiality in two major areas. The Solidity Contracts team and the Zama team are commended for addressing user needs by incorporating confidentiality into these protocols and their 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, the code was found to grant permissions in a granular yet strict way, allowing only the necessary parties to be able to decrypt and re-encrypt 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. The audit team is glad to have collaborated with them on working towards this milestone.