Alethea AI

Smart Contracts Audit

by

Sheraz Arshad

Summary

This report represents the code review of the three smart contracts from `Alethea AI`, along with all the contracts inherited and libraries used by them. The name of the smart contracts are `PolygonAliERC20v2.sol`, `NFTFactory.sol` and `WhitelabelNFT.sol`. A static (using Slither) and manual review of the contacts was performed. No critical vulnerabilities were found in the code review. The smart contracts' code can be improved by incorporating the suggestions from following findings, most of which point to the areas of smart contract that has room for gas optimization.
The commit hash used for the audit is 34801d72f3e60728586ded4085efacc827ea8ca7

PolygonAliERC20v2

The `PolygonAliERC20v2` extends `AliERC20v2Base` contract and adds functions to allow bridging of `ALI` ERC20 tokens to and from Polygon sidechain. The `AliERC20v2Base` contract is the utility ERC20 token contract from Alethea AI.

Apart from the unlocked compiler version, no vulnerabilities are found in this contract.

AliERC20v2Base

The `AliERC20v2Base` is the utility ERC20 token contract from Alethea AI. This contract adds a number features on the top of standard ERC20 functionality. It inherits from `ERC1363`, `EIP2612`, `EIP3009` interfaces and `AccessControl` contract.

The implementation of `ERC1363` interface adds functionality of allowing transfer or approval recipient to be notified of it with a message call within a single transaction.

The implementation of `EIP2612` interface adds functionality of `permit` using typed signed `ERC-712` compatible signatures. This allows the gasless token approval for message signer and as well as performing of approval and transfer within a single transaction.

The implementation of `EIP3009` interface adds functionality of tranfering and receiving tokens using authorized or signed messages, where a tokens sender can sign a message to send tokens with anyone paying for the gas or the tokens recipient can take signed message from the sender and execute the transaction paying for the gas themself.

The inheriting of contract `AccessControl` adds functionality of managing of privileged roles that are allowed to enable or disable key features of the contract tied to their roles.

The contract also implements Compound DAO inspired votes delegating mechanism where tokens balance can be used to vote on proposals.

AEB-01

The comments on lines `L44-L46` mention that the `mint` functionality will be disabled by revoking `TOKEN_CREATOR` permission once the contract is deployed with no further minting possible. This behaviour should be coded within the smart contract to avoid any uncertain situation where the behaviour mentioned in the aforementioned comments is not properly followed. Either a maximum supply constant can be introduced ensuring total supply never exceeds `10B ALI` or a check should be introduced in `mint` function that disallows calling it once the contract is deployed and its `constructor` has run.

AEB-02

The lines `L1840`, `L1845` and `L1847` read length of auxillary data structure array directly from storage. This results in increased execution gas cost since reading from storage is expensive than reading from stack or memory. The recommendation is to store array length in a local variable and utilize it on the aforementioned lines.

AEB-03

The lines `L1319` and `L1323` read `tokenBalances[_from]` directly from contract's storage, which is expensive than reading from stack or memory. The recommendation is to make use of local variable to avoid additional gas cost.

AEB-04

The code blocks from `L1025-L1047` and `L1282-L1305` are identical. The can extracted to a private function to reduce the bytecode footprint of the contract. A reduced bytecode footprint will result in reduced deployment cost against the contract.

AEB-05

The line `L1876`, `L1883`, `L1885` and `L1889` read length of auxillary data structure array directly from storage. This results in increased execution gas cost since reading from storage is expensive than reading from stack or memory. The recommendation is to store array length in a local variable and utilize it on the aforementioned lines.

AEB-06

The contract has most of its functionality such as `burn`, `transfer` and `transferFrom` controlled by special roles bearing addresses that can enable or disable the features that they have the priviledge for. This will result in strong centralization of the token contract if those roles are ever assigned outisde the DAO based contracts. The recommendation is to put in place a mechanism to ensure only a DAO based contracts can have those roles. One such example is a proposal based DAO contract that assigns or revokes roles through voting.

NFTFactory

The `NFTFactory` is a helper contract that allows minting on arbitray Mintable `ERC-721` compatible contracts. It inherits from `AccessControl` contract to manage privileged roles. Only the privileged role of `ROLE_FACTORY_MINTER` is allowed to mint tokens on behalf of recipient by either making the transaction itself or through issuing signed messages. The contract utilizes `EIP-3009` inspired arbitrary nonce tracking mechanism for the signed messages from `ROLE_FACTORY_MINTER` role bearing addresses.

NFF-01

The check `_tokenId != 0` on `L168` ensures the token with id `0` cannot be minted but it limits the functionality on NFT contract where the minting is to be performed if it allows minting the token with id `0` as is the case with `TinyERC721` contract. The recommendation is to either remove this check or move into the NFT contract e.g. TinyERC721 for better code legibility.

WhitelabelNFT

The `WhitelabelNFT` contract inherits from `RoyalERC721` contract with little functionality of its own other than tracking the contract's id and calling inherited contract's constructor.

No vulnerabilities are found in this contract.

RoyalERC721

The `RoyalERC721` is royalty supporting `ERC-721` contract that inherits from `EIP2981` interface and `TinyERC721` contract.

The implementation `EIP2981` interface adds functionality of returning royalty information such as royalty fee and fee recipient for the NFT markets.

`TinyERC721` contract is the concrete implementation of the NFT itself.

ROE-01

The function `setRoyaltyInfo` on `L153` allows the `ROLE_ROYALTY_MANAGER` role bearing address to change royalty fee on its discretion, where they can set it to take all of the sale amount from the purchase as royalty fee by setting the royalty fee to 100%. The recommendation is to set a cap on how much maximum royalty fee can be taken from a purchase.

TinyERC721

The contract `TinyERC721` is a novel implementation of enumerable `ERC-721` compatible contract. Other than the extended implementations of `ERC721`, this contract inherits from `AccessControl` contract to manage privileged roles. It implements `permit` functionality using typed signed ERC-712 compatible signatures. This allows the gasless token approval for message signer and as well as performing of approval and transfer within a single transaction.

TEC-01

The constant declared on `L209` is never used in the contract and should be removed for code legibility.

TEC-02

The contract has most of its functionality such as `burn`, `transfer` and `transferFrom` controlled by special roles bearing addresses that can enable or disable the features that they have the priviledge for. This will result in strong centralization of the token contract if those roles are ever assigned outisde the DAO based contracts. The recommendation is to put in place a mechanism to ensure only a DAO based contracts can have those roles. One such example is a proposal based DAO contract that assigns or revokes roles through voting.

TEC-03

Refer to `NFF-01`.

AccessControl

The `AccessControl` allows managing of privileged roles in the contracts inheriting from it.

ACC-01

The lines `L145` and `L148` read `userRoles[operator]` directly from contract's storage, which is expensive than reading from stack or memory. The recommendation is to make use of local variable to avoid additional gas cost.

General

The general recommendation is to lock the compiler version across all contracts. An unlocked compiler version allows compiling the contract at the specified version or above it. Different compilers can generate different bytecode and prone to compiler specific bugs. The recommendation is to lock the compiler version on the contract, so it can be compiled only with the specific version of the compiler.