Alethea AI | Smart Contract Audit | Polygon NFT & Web3auth

tags: `alethea , audit`

Latest revision: December 3rd
Miguel Palhas **mpalhas@gmail.com** (mailto:mpalhas@gmail.com)

Table of Contents

Overview

A permissioned NFT minting factory in Solidity

Dates

- **August 14th**: Start date
- **August 15th**: First report

Process

This document, and all suggestions detailed here, is meant to be scrutinized and discussed between both parties, in an ongoing collaborative process after the first report delivery. Each suggestion may not be needed due to contextual reasons, or limited understanding of external scope by the auditor.

Coverage

The following repositories were considered in-scope for the review:

- **https://github.com/AletheaAI/alethea-contracts** (https://github.com/AletheaAI/alethea-contracts) (revision `a475730`)
- **https://github.com/Alethea-FullStack/frontend-phase1** (https://github.com/Alethea-FullStack/frontend-phase1) (revision `98ffee5`)

In particular, this audit focuses solely on the upcoming launch on the Polygon chain, which includes the `PolygonAliERC20v2` contract, and the accompanying frontend code.

Areas of Concern

The investigation focused on the following:

- Looking for attack vectors that could impact the intended behaviour of the smart contract
- Checking test coverage, particularly for potentially dangerous scenarios and edge-cases
- Interaction with 3rd party contracts
- Use of solidity best practices
- Permissions and roles after deploy script is executed

Findings

Each issue has an assigned severity:

- **Informational**
- **Minor** issues are subjective in nature. They are typically

suggestions around best
practices or readability. Code maintainers should use their judgement as to
whether to address such issues.

- **Gas**
  issues are related to gas optimizations.
- **Medium**
  issues are objective in nature but are not security vulnerabilities. These
  should be addressed unless there is a clear reason not to.
- **High**
  issues are security vulnerabilities that may
  not be directly exploitable or may require certain conditions in order to be
  exploited. All major issues should be addressed.
- **Critical** issues are directly exploitable security vulnerabilities that need to be fixed.

Progress

This table helps tracks the auditor's progress through each section of the
codebase, and is not indicative of final results.

`PolygonAliERC20v2.sol`

identifier	progress
constructor	✅
deposit	✅
withdraw	✅

`web3Auth`

Findings Summary

Findings are listed in chronological order of discovery.

title	severity
1. Consider using OpenZeppelin's `EIP712` library	Informational
2. Hardcoded references to mumbai testnet and goerli	Medium
3. 3. Potentially insecure middleware authentication	High

Detailed Findings

1. Unnecessary minting of initial supply

Severity: Informational

The `PolygonAliERC20v2` contract inherits from `AliERC20v2`, which was originally meant for Ethereum mainnet.

One leftover from this is the fact that it mints a hardcoded initial supply of tokens.

This is an unecessary step, albeit not very dangerous, since the supply is imediatelly burned on the deploy script.

2. Hardcoded references to mumbai testnet and goerli

Severity: Medium

The web3 client on `web3Auth` is connected to Polygon's mumbai and ETH Goerli networks:

```
const web3 = new Web3(
  new HDWalletProvider(
    TREASURY_KEY,
    'https://matic-mumbai.chainstacklabs.com',
  ),
);
```

Other references also exist on various `console.log` statements.

In addition, Web3auth connection is hardcoded to the goerli network:

```
const web3auth = new Web3Auth({
  clientId,
  chainConfig: {
    chainNamespace: CHAIN_NAMESPACES.EIP155,
    chainId: '0x5',
    rpcTarget: 'https://rpc.ankr.com/eth_goerli',
  },
});
    web3auth.init({ network: 'testnet' });
```

Recomendation: Use an environment variable to distinguish testnet from mainnet environments.

3. Potentially insecure middleware authentication

Severity: High

The authentication to the web3Auth middleware is apparently done via a hardcoded `SERVICE_KEY` environment variable.

This is a rather insecure practice, since a leak of this secret key may open up access to the entire service to 3rd parties. Since this microservice handles real tokens on behalf of users, access to it means loss of user funds.

A more sensible security practice is to use per-request HTTP signature headers, which are more cryptographically secure, and scoped to each specific request. A leak of a single request does not compromise general access to the microservice.

Note: The severity of this depends highly on the underlying infrastructure. Without knowledge of the deployment setup, it's impossible to properly judge the risk.