Smart Contract Audit Report
for
Arts Dao - Ethernal Gates

by

Rtility

Version 1.0 | April 4, 2022

Rtility

Email: rtility.io@gmail.com

Introduction:

In this report, we’ll cover the following topics:

Disclaimer:

The audit makes no statements or warranties about the utility of the code, the safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about the fitness of the contracts to the purpose or their bug-free status. The audit documentation is for discussion purposes only.

Overview:

We received a single file that combined all the library and other abstract contracts. At the same time, the project wasn't audit-ready, causing some functions not to be commented on, but overall, it was easy to understand.

We extracted the main contract from that file and made a single file; all the lines we mention in this report are based on our file; you can find the file here.
The project is an ERC721 (NFT) token, implemented using ERC721A, an improved implementation of the ERC721 standard That supports minting multiple tokens for the cost close to minting one, an excellent practice nowadays.

We conclude that the project was developed using Remix IDE, a good IDE for debugging and interacting directly with live networks and quick prototyping. However, the alternative js ecosystems like hardhat are more suitable for this project.

The project doesn't have any test suit, which is a bad practice for developing a smart contract. Having close to 100% test coverage guarantees that your code works as intended and empowers us, auditors, to spend more time identifying security issues rather than functional bugs. However, the contract doesn't have any bugs in functionality, judging by our in-depth line-by-line inspection.

Some minor style nits and optimization improvements could be done to be more gas-efficient for runtime users. We provided our potential gains in the line-by-line review, and the style fixes are reported later using slither, a static analyzer for solidity.

Known Vulnerability Analysis:

Reentrancy attack

From hackernoon:

“The Reentrancy attack is one of the most destructive attacks in the Solidity smart contract. A reentrancy attack occurs when a function makes an external call to another untrusted contract. Then the untrusted contract makes a recursive call back to the original function in an attempt to drain funds.”

In this project, two functions make an external call:

The first is in the `withdraw` function at #L73; because this function is `onlyOwner`, there are no concerns for reentrancy here.

The second call is in the ERC721A internal `_mint` function. This method added a reentrancy guard on the v3.0.0 release (See here); You are safe because you are using a newer version than that.

So, overall you are safe from Reentrancy.

Over and underflows

From hackernoon:

“In simple words, overflow is a situation when uint (unsigned integer) reaches its byte size. Then the next element added will return the first variable element. In the case of underflow, if you subtract 1 from a uint8 that is 0, it will change the value to 255.”

Since Solidity 0.8.0, the overflow/underflow check is implemented on the language level by default and could be ignored using the `unchecked` wrapper.

You are using Solidity 0.8.11, and there isn’t any `unchecked` wrapper in your base contract; therefore, you are safe from over and underflow attacks.

Features:

The contract offers a good functionality intended in the checklist provided by the team:

Features requested in the checklist:

- ✅ Using OpenZeppelin Ownable, the contract owner can transfer ownership to others. #L4
- ✅ The contract owner can mint NFTs for free up to currentSupply. #L77-L80
  However, we advise limiting how many tokens can be minted by owners.
- ✅ The contract owner can airdrop NFTs up to currentSupply. #L82-L92
- ✅ The contract owner can withdraw ETH. #L72-75
  The function is well written using the `call` method.
- ✅ The contract owner can increase the supply up to the MAX_SUPPLY constant initialized to 6,000. #L67-70
- ⚠️ The contract owner can change the sale status to one of these phases: #L50-52
  - OFF
  - INVESTOR
  - VIP
  - WL
  - PUBLIC

  Note that two different phases cant be active at the same time.

- ❗ All tokens (apart from airdrop) are minted to`msg.sender`, the user **or the contract** who calls the function.
  Note that the user could receive the NFT at a different address than the one they pay the mint fee using another contract (contract mint). This issue is addressed in the line-by-line report, and a suggestion is proposed. See here

Additional features:

- ⚠️ The contract owner can change presale and public prices at #L58-64. From a technical point of view, See here.
- ✅ Whitelisting is done using an off-chain whitelist and on-chain validation using the Merkle tree approach, which is common among NFT projects and very gas efficient for owners against using on-chain whitelisting.
  The alternative way for off-chain whitelist and on-chain validation is using Ethereum signed message with the ECDSA signature that slightly uses less gas, but the Merkle tree is fine.

Vulnerabilities:

The severity levels of the issues are described as:

- Critical: The issues will result in asset loss or data manipulations.
- High ❗: The issue will seriously affect the correctness of the business model.
- Medium ⚠️: The issue is still important to fix but not practical to exploit.
- Low: The issue is mainly related to outdated, unused code snippets that don't damage the contract.
- Informational 📝: The issue is mainly related to code style, informational statement, or gas optimization (♻️) and is not mandatory to be fixed.

Total	Critical	High ❗	Medium ⚠️	Low	Info 📝 ♻️
25	0	1	4	0	20

No critical vulnerability is found.

The one high issue that we came across is the one for contract minting addressed earlier in this report in the features section.

All of the medium issues are related to not having a batch size for minting. A detailed description is reported here.

For the 20 nits and gas optimization that we found, please check the line-by-line inspection on GitHub here. For being focused on optimization, there are some styling issues that we are not addressed in the review, but all of them can be found in the slither report that we will provide you later on in this report.

Manual Analysis: Line-by-line inspection

Our line-by-line report is a review of a GitHub PR containing your base code located here.
For any further discussions, please refer to the GitHub PR.

Automated Analysis:

We use slither, a static solidity analyzer, for automated analysis. The full report of slither with our comments can be found here. There aren’t any high severity issues, but some style nits are worth mentioning.