byterocket

async.

AsyncArt v2

May 26, 2020

1. Preface

The team of **AsyncArt** contracted us to conduct a software audit of their developed smart contracts written in Solidity. AsyncArt is a project asking "what does art look like when it can be programmed?" while exploring new ways of visualizing and implementing the idea of art. They are separating the art into "master" and "layer". While the "master" is the piece of art itself, a "layer" represents a single part of the artwork. Masters as well as layers are tokenized on the Ethereum blockchain and can be bought and sold using Ether.

The team of AsyncArt is planning to upgrade their current version of the deployed contracts with an upgrade. These new, updated contracts are what is being reviewed.

The following services to be provided were defined:

- Manual code review
- Protocol/Logic review and analysis (including a search for vulnerabilities)
- Written summary of all the findings and suggestions on how to remedy them (including a Zoom call to discuss the findings and suggestions)
- Final review of the code once the findings have been resolved

We gained access to the code via the public GitHub repository via https://github.com/asyncart/async-contracts/tree/upgrade/contracts. The state of the code that has been reviewed was last changed on the 20th of May 2020 at 02:52 AM CEST (commit hash 1bbca6bfe1a171f1bb8369ff129d5aac234a6664).

2. Manual Code Review

We conducted a manual code review, where we focussed on the two main smart contracts as instructed by the AsyncArt team: "AsyncArtwork_v2.sol" and "TokenUpgrader.sol". For a description of the functionalities of these contracts refer to section 3.

The code of these contracts has been written according to the latest standards used within the Ethereum community and best practice of the Solidity community. The naming of variables is logical and comprehensible, which results in the contract being easy to understand. As the AsyncArt project is a decentralized and open-source project, these are important factors.

The comments in the code help to understand the idea behind the functions and are generally well done. The comments are also used to explain certain aspects of the architecture and implementation choices.
On the code level, we did **not find any critical bugs or flaws**. We did however find seven flaws with none or low severity that we listed below. An additional double-check with two automated reviewing tools (one of them being the paid version of MythX) also did not find any bugs. The automated tools merely noted that there are loops within the code that could lead to excess gas usage if the underlying data structures were growing unbounded. We assume that the developers are aware of this.

2.1. Bugs and Flaws (AsyncArtwork_v2.sol)

A) Line 314-319 [LOW SEVERITY]

```
for (uint256 i = 0; i < additionalCollaborators.length; i++) {
```

```
    // can't provide burn address as collaborator
    require(additionalCollaborators[i] != address(0));
    uniqueTokenCreators[controlTokenId].push(additionalCollaborators[i]);
  }
```

This section from “setupControlToken()” does not contain the same checks that are applied during the “mintArtwork()” function. The “uniqueTokenCreators” array will contain double entries of the same address if it is called in a specific way. As this can and probably will be prevented from the frontend, we only would suggest fixing this if that is not the case.

B) Line 446 [NO SEVERITY]

```
// Allows the buyer to specify a minimum remaining uses they'll accept
function takeBuyPrice(uint256 tokenId, int256 expectedRemainingUpdates) external payable {
```

The comment states that the amount of minimum remaining uses can be specified by the user, however, the function only executed if the remaining amount is exactly as stated.

```
require(controlTokenMapping[tokenId].numRemainingUpdates == expectedRemainingUpdates);
```

We suggest to either change the comment if this behavior is desired - or the require if this is not desired.

C) Line 599 [NO SEVERITY]

```
int256 previousValue = lever.currentValue;
```

The variable “previousValue” is not necessary since it is never used again outside of this scope. Instead, it could be replaced with a modified version of line 605 like this:

```
previousValues[i] = lever.currentValue;
```

D) Line 155 [NO SEVERITY]

```
mapping(uint256 => ControlToken) controlTokenMapping;
```

There is no visibility set, such that it will default to internal. We suggest to either declare it as private or public.

E) Line 20-134 [NO SEVERITY]

All of the events don’t contain any indexed properties. It’s up to the team of AsyncArt whether this is okay or not, but it might lead to problems in the future if certain filtered web3 calls need to be made.

F) Line 472-478 [LOW SEVERITY]

The function “distributeFunds()” is dividing a provided amount of Ether into equal parts and transfers it to the provided addresses. In the case of a division with a remainder this remainder will remain inaccessible in the contract. This could, for example, happen if the result of the division in line 473 would result in 33,3, but since integers are used, the used value will be 33, leaving a remainder of 0,3.

Since ETH is denominated in 10^18 Wei, the impact of this remainder will be negligible, since it will be in the area of (at current ETH price of about 200 USD) 10-16 USD. We’re just noting it for the sake of completeness. It could be mitigated by returning “creatorShare.mul(creators.length)” at the end of “distributeFunds()” and then use this to calculate the “real” remaining amount to be sent in “safeFundsTransfer()”.

G) Line 644 [LOW SEVERITY]

Currently it is not recommended to specify a fixed gas-amount in a “call.value” since it won’t allow smart contract wallets to receive ether. Instead, the currently recommended way to send ether is to use

```
(bool success, ) = msg.sender.call.value(amount)()
```

Using this method, it is very important to do this as late as possible within the execution of the contract to prevent reentrancy attacks. Since you allow users to manually claim their failed transfers with a special function, no user funds will be locked due to this, and users using smart contract wallets can use this function to receive their ether.

As the current implementation might be annoying for certain users, it might be worth thinking about changing the current implementation to the proposed solution stated above (or also used in line 637). In that case it has to be of utmost importance to ensure not allowing any reentrancy attacks while doing so.

Since we are sure that the decision to include the 2300-gas-call was done on purpose and has been well-thought-out, we won’t recommend any changes here. We just want to state the current recommended way to transfer ether for completeness.

2.2. Bugs and Flaws (TokenUpgrader.sol)

We did not find any bugs or flaws in TokenUpgrader.sol and the corresponding “upgradeV1Token()” function in

3. Protocol/Logic Review

Part of the audit was also an analysis of the protocol and its' logic, together with an analysis of whether this protocol works as intended or contains any logical bugs. Starting with the description of the functionality of the protocol in section 3.1 and 3.2, ending with a breakdown of our findings in section 3.3.

3.1. Functionality Descriptions

Generally, the functionalities of the contracts can be divided into three sections:

- Platform-only functions
- Buy-Sell-related functions
- Token-related functions

While the platform-only functions can only be called from the platforms' address, the other functions may be called by anyone. We omit a detailed list of all functions and only list those that we consider relevant for the purpose of the protocol analysis.

Platform-only functions

- WhiteListTokenForCreator
- Allows an address to mint one master token and X layer tokens (where X can be defined by the platform)
- WaiveFirstSaleRequirement
- Waive the right of the platform to receive their 10% fee for the first sale, instead, they'll immediately switch to receiving just 1%. The artist receives 99% of the first sale.
- Update Fees (updatePlatformSalePercentage)
- Set new minimum bid percentage (updateMinimumBidIncreasePercent, updateArtistSecondSalePercentage)
- Change a token URI (updateTokenURI)
- We assume that this function only exists in order to help artists to fix/set a proper URI, without intending to abuse it in any way. If that is not it's intended use-case, we might need to reevaluate.
- Lock a token URI (lockTokenURI)
- Transfer a v1 token to v2 (upgradeV1Token)

Buy-Sell-related functions

- Bid
- Places a bid of a certain amount on a certain token
- WithdrawBid
- Removes a users' bid from a token
- AcceptBid
- Accepts a users' bid on a token (as the owner of the token)
- MakeBuyPrice
- Set a fixed buy price that anyone can immediately buy the token for (as the owner of the token)
- AcceptBuyPrice
- Buy a token for a fixed buy price

Token-related functions

- MintArtwork
- Mints a master token as a whitelisted artist, also allocates the layers (can also directly be given to other addresses)
- MintControlToken

- Mints a layer token and sets its' levers (options to control it), together with a list of artists that might have collaborated with the minting artist
- Also allows an artist to limit the number of uses
- UseControlToken
- Change the levers/values of a layer token as the owner or someone who was given permission by the owner
- GrantControlPermission
- Give someone else permission over the control of the users owned layer token

3.2. Protocol Logic

While the functionality description covers most of the logic, there are a few things that we want to highlight in order to understand the protocol in total.

Buy mechanism

There are two ways for a token to be sold: either through an auction-like mechanism where users bid a price that the owner can accept after it has reached the desired amount. The other option for the owner is to set a price that the token can instantly be bought at by anyone.

Fee System

At the time of this audit, the current fee system is implemented (but can be changed by the platform at any time):

3.3. Vulnerabilities and Flaws

During our analysis of the protocol and its logic we did **not find any bugs or flaws**. Functions involving monetary aspects (like transferring Ether) have been separated from other art- or control-based functions which further reduces the risks of the involved funds. During our tests, we were **not** able to maliciously game the protocol.

4. Summary

During our code review (which was done manual as well as automated) we found 7 bugs and/or flaws, with **none of them being critical**. All of the found flaws were of low or no severity. Since none of these bugs pose any risks to the use of the protocol or the funds of the users, we will leave it to the AsyncArt team to decide whether they want to address them or not.

In our analysis of the protocol and the logic of the architecture, we did not find any bugs or flaws and we were not able to maliciously game the protocol through the tests that we did.

Overall we see that the developers adhered to the best practices of Solidity and did a good job implementing their idea.

5. Update on the 04th of June 2020

Since we sent our report to the AsyncArt team, the findings have been discussed in a bi-lateral meeting. All of our found flaws have been addressed:

A) Line 314-319 [LOW SEVERITY]

The developers are aware of this and it is an intended feature. The naming of the variable might not be ideal, but given that the idea of the developers is, that the percentage of the paid amount can be changed through this mechanism to favor certain collaborators, we don't consider this a flaw anymore.

B) Line 446 [NO SEVERITY]

The wording has been changed in GitHub commit b9ca792c07d85d8dba7cdc3940485b997f128b7c.

Before

```
// Allows the buyer to specify a minimum remaining uses they'll accept
```

After

```
// Allows the buyer to specify an expected remaining uses they'll accept
```

C) Line 599 [NO SEVERITY]

This flaw has been adresses in GitHub commit b9ca792c07d85d8dba7cdc3940485b997f128b7c.

Before

```
// grab previous value for the event emit
int256 previousValue = lever.currentValue;

// Update token current value
lever.currentValue = newValues[i];

// collect the previous lever values for the event emit below
previousValues[i] = previousValue;
```

After

```
// grab previous value for the event emit
previousValues[i] = lever.currentValue;

// Update token current value
lever.currentValue = newValues[i];
```

We **don't** think that this change has introduced any new risks or flaws.

D) Line 155 [NO SEVERITY]

This flaw has been adresses in GitHub commit 9bf3dd685c871d76fa47cbb32c51585d8e611035.

Before

```
mapping(uint256 => ControlToken) controlTokenMapping;
```

After

```
mapping(uint256 => ControlToken) public controlTokenMapping;
```

We **don't** think that this change has introduced any new risks or flaws.

E) Line 20-134 [NO SEVERITY]

The developers explained that they intend to use "The Graph" to query data from the blockchain, such that indexed events are not needed in this case.

F) Line 472-478 [LOW SEVERITY]

The developers acknowledge that our findings are true, but we both agree that changing this does not provide any benefits as the remainders value is negligble.

G) Line 644 [LOW SEVERITY]

The developers explained, that this is an intended behavior by the smart contract. They want to make the automated transfer functionality to be as safe as possible, without introducing any risks through a possible abuse of the gas system. They keep the current implementation, which is perfectly fine since they allow smart contract wallet users to withdraw their funds manually in case of any problems. No user funds are in danger, such that we don't consider our finding a flaw anymore.

Disclaimer

accuracy of the information provided. The auditors hereby disclaim, and each client or user of this audit report hereby waives, releases and holds all auditors harmless from, any and all liability, damage, expense, or harm (actual, threatened, or claimed) from such use.

Download the Report

Stored on IPFS

We store our public audit reports on IPFS; a peer-to-peer network called the "**I**nter **P**lanetary **F**ile **S**ystem". This allows us to store our reports in a distributed network instead of just a single server, so even if our website is down, every report is still available.

Learn more about IPFS →

Signed On-Chain

The IPFS Hash, a unique identifier of the report, is signed on-chain by both the client and us to prove that both sides have approved this audit report. This signing mechanism allows users to verify that neither side has faked or tampered with the audit.

Check the Signatures →