byterocket

REALITY CARDS

RealityCards v1

May 19, 2020

1. Preface

The team of **RealityCards** contracted us to conduct a software audit of their developed smart contracts written in Solidity. RealityCards is a project combining the idea of prediction markets, non-fungible tokens and harberger taxes, where at the end of the season/event all holders of the winning token will receive a split of the total rental payments in proportion to how long they have held the token through staking a stable coin called DAI on the Ethereum blockchain.

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 of 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/RealityCards/RealityCards-Contracts/blob/master/contracts/RealityCards.sol. The state of the code that has been reviewed was last changed on the 17th of May 2020 at 12:11 AM CEST (commit hash aad8ea70696d848e2fcb55b7932c7ba37b8f239e).

2. Manual Code Review

We conducted a manual code review, where we focussed on the main smart contract as instructed by the RealityCards team: "RealityCards.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 RealityCards 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 bugs** or flaws. An additional double check with two automated reviewing tools (one of them being MythX) also did not find any bugs.

2.1. Other Findings

While we did not find any bugs or flaws, we want to note the following:

Possibly unnecessary variable "nftMintCount"

The variable "nftMintCount" introduced in line 26 is most probably not necessary, since ERC721 does provide a totalSupply function that can be used in exchange.

Possibly unnecessary function "userRemainingDeposit()"

This function can probably be combined with "*currentOwnerRemainingDeposit()*" to always return the remaining deposit, no matter whether an owner or past owner/user calls it.

Possibly unnecessary check for zero

In lines 209 - 211 there is a zero check that is probably not necessary since pps can never be zero if everything is implemented correctly. The minimum price of 0.01 DAI (1016 attoDai) defined in line 354 means that the pps couldn't be lower than 115.740.740.740 pps (= 1016 / (24x60x60)).

3. Protocol/Logic Review

The description and specification of the protocol was provided to us via a GitHub link. We conclude that the present implementation complies with the specification and depicts it.

3.1. Functionality Descriptions

A comprehensive description of all the functions and their respective functionalities is given in this Google Doc. We validated the code and logic against this specification.

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.

Tokens, Users and the System

The variables and connections within the protocol are structured in basically three layers: tokens, users and the system itself. Each of them have certain properties that are attached to it.

Starting from the top, we have a representation of a token, which is implemented according to the ERC721 standard (overwriting the standard transfer functions such that only the contract may send/control them according to the protocol). Below are the variables that are attached to each of the tokens, with the ones in blue only being relevant for the front-end and are not being used within the protocol logic.

One level deeper we have each of the users, where we only store the total amount of rent that they paid as well as an indicator whether the user already withdrew their winnings or not.

At the lowest level we have the system itself: the contract. Some of these variables are generally necessary to ensure the safety of the system (like the minted tokens variable or the limiter for iterations. The rest are general values, which are used for all other things.

States of the System

The protocol itself is divided into four states that define which functions can be called. The transition between states is done via a function that increments the state variable which can be controlled by any user that wishes to do so except for the first transition, where only the contract owner is allowed to do so. This is not a problem since no user is able to interact with the contract or deposit anything into it before the second state OPEN is enacted.

A function called "circuitBreaker()" allows the owner to transition the protocol into the WITHDRAW state immediately, more on this in the next section.

Calculation of the Time Held

A specialty of this protocol is the way that the winnings are split and distributed after the event has ended. In RealityCards, the only factor that is relevant here is the time that a user held the winning token. If two users for example both held a token for 50% of the time but paid different prices/rent, they will still both receive the same payout. This is an intended behaviour. An example calculation:

Payout

User X receives 36 DAI (54 total DAI / 6 days total x 4 days held), while users Y and Z both receive 9 DAI (since both held the token for 1 day each). User Y paid 24 DAI in rent while user Z paid 10 DAI.

3.3. Vulnerabilities and Flaws

A) Possible abuse by owner address

While the whole protocol is built in a trustless way, there is one function that potentially allows the owner to act maliciously: “circuitBreaker()”. A function that allows the owner to immediately transfer the protocol into the “withdraw state”, skipping the other states and the consideration of the prediction markets outcome. The following scenario would be possible:

- Owner X deploys the contract and initializes the NFT tokens T1 and T2 correctly
- X buys token T1, while user A buys token T2
- As soon as it looks like token T2 wins, X calls “*circuitBreaker()*”
- X and A receive their rent and deposit back.

While nobody wins, X just prevented himself from losing something and A from winning something. Even if a mechanism would be in place that doesn’t allow the owner's address to participate in the token mechanics, there is no identity system that would prevent them from creating another account.

We acknowledge that the team of RealityCards most likely has no bad intentions. There is no risk for any user funds, since the worst outcome would be a missed profit. But since Ethereum’s community values the ability to govern things in a decentralized fashion instead of being governed by a centralized administrator, we would suggest to change this in the future if possible.

Possible mitigation

One way to mitigate this would be the implementation of a voting scheme. If the users are willing to enter the WITHDRAW state, they should be allowed to do so, instead of relying on a benevolent owner. Maybe a simple voting scheme that allows any user that has a “*collectedPerUser*” of greater than 0 would be a solution.

We acknowledge that this may involve other risk factors, since basically all holders of the non-winning tokens are then incentivized to vote in favor of this in order not to lose any money. We merely want to state that the current solution might not be the perfect one.

B) Array to store past owners

While an array is probably the easiest solution to implement a list of past owners, it introduces problems as well. In this case, the ownership of a token reverts to the last owner if the current one runs out of deposited money. If this owner also does not have any money deposited anymore it will be transferred to the owner before that and so forth. This can potentially lead to a long array that can not be processed within one transaction call, which is known to the developer(s). The current solution is a variable that caps the iterations at 10 per call, where a temporary owner is selected if no eligible owner was found after 10 iterations. This solution is especially susceptible in the early stages of the project, since it might happen that the protocol does not get called often enough to transfer ownership to the right owner within a reasonable time.

Possible mitigation

The use of a doubly linked list instead of an array would probably solve this problem, since every user who withdraws their full deposit would be able to (efficiently) remove themselves from the list, such that the “*_revertToPreviousOwner()*” function could always rely on the previous node within the doubly linked list to be a valid one.

Since the current implementation does not pose any immediate risks for the users or their users, we classify this flaw as low risk.

C) Intermediate time is not considered

Consider the following example:

User A is the current owner of a token (with a sufficient deposit). User B buys this token (at a higher price, with a deposit that will last them a day). After one day, the token will revert to user A, since user B ran out of money to pay the rent. However, this will only happen if someone calls a function that triggers the rent collection mechanism. If that is not the case, user B might be owning the token for longer. While there is a mechanism that only counts the actual time that user B was able to pay with his deposit, user A does not receive credit for the intermediate time. If, in our example, user C calls a contract function that triggers the rent collection for the token after two days, user B will only receive one day as “time held”, while user A will not be credited with the remaining day that they would have held the token. Since they are not paying any rent during that time there is no immediate harm done but this could probably be improved.

Possible mitigation

We know that the implementation of a mechanism that also counts the time that user A would have gotten if they would have possessed the token (in case someone triggered it) introduced more complexity and probably also increases the gas cost of the rent collection. We merely want to note it, since we don’t think that this is an ideal implementation.

Possible solutions that solve the issue without any further calculations would be the introduction of a little reward for anyone calling the “*collectRentAllTokens()*” function (maybe once every X hours), comparable to a “mining” fee.

In practice, it is likely that the owners/developers of RealityCards would probably call this function regularly in order to prevent this from happening until enough users are using their product, where this issue might not be relevant anymore, since each token is interacted with frequently.

D) External risk through prediction market

As the review of the external prediction market was not within the scope of our review, we can't assess the risk that it might pose to the protocol. We are aware that the used provider for the prediction market uses a reasonable approach based on the principles of community-vetting. If the prediction market provided wrong information, the RealityCards protocol wouldn't work as it relies on and trusts its outcome. Malicious actors of the prediction market could potentially corrupt the RealityCards protocol by abusing it.

As long as the prediction market can be trusted, we don't see any issues here. Maybe an emergency-vote like solution can mitigate some of these risks in the future.

4. Summary

Overall the smart contract is very well written and cleverly thought out. During the manual code review we did not find any bugs or flaws. Also our automated tools did not find anything. The comments of the developers are very helpful and well-done.

Our protocol and logic analysis did show four possible flaws. None of them are posing a threat to the users funds, but they also should not be ignored. In particular, a solution to **3.3B** ("Array to store past owners") would significantly increase the elegance of the protocol.

5. Update on the 26th of May 2020

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

A) Possible abuse by owner address

The ability of the owner to call the circuitBreaker function was removed in Github commit 041d98bf4f390d122cb4be2c62ab8a65b3c46bc4.

Before

```
require(msg.sender == owner() || now > (marketExpectedResolutionTime + 4 weeks), "Not owner or too early");
```

After

```
require(now > (marketExpectedResolutionTime + 4 weeks), "Too early");
```

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

B) Array to store past owners

Our suggestion has been acknowledged, however, both sides think that the complexity of this possible change is not necessary as the current approach works as intended. Also it is not sure whether a doubly linked list solves the problem.

C) Intermediate time is not considered

This suggestion has been acknowledged and might be addressed in a later upgrade.

D) External risk through prediction market

The developer is aware that this is a general risk. The risk is considered to be very low.

2.1 Other Findings - Possibly unnecessary check for zero

The unnecessary check has been removed in GitHub commit 8a1223980eb75cad4a0d5ed784545776b1586037

Before

```
if (pps == 0) {
    return now;
} else {
    return now + currentOwnerRemainingDeposit(_tokenId).div(pps);
}
```

After

```
return now + currentOwnerRemainingDeposit(_tokenId).div(pps);
```

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

2.1 Other Findings - Possibly unnecessary function "userRemainingDeposit()"

The developer explained that this function is necessary for front-end use, so that it needs to exist.

New Finding: Zero winners edge case

Since our audit another bug was found that occurs during an edge case where the winning token was never bought by an user. The rent that has been accrued through other tokens would have been locked in the contract. It was fixed in GitHub commit 79a53c772780f7b881d06eeb38ff52ff98341c3d.

Before

```
if (winningOutcome !=  ((2**256)-1)) {
```

After

```
if (winningOutcome !=  ((2**256)-1) && totalTimeHeld[winningOutcome] > 0) {
```

Since "totalTimeHeld" is the most reliable approach to define whether a token has been bought or not we think that this solution is appropriate. We **don't** think that this change has introduced any new risks or flaws.

New Change: Can now transfer token in withdraw state

The developer wanted to allow users to transfer their token after the market has been resolved, since the last owner otherwise would have to keep the token in their wallet forever. Since some users might not like this, he changed it in GitHub commits 62277a097a2d35be6e28d0e78b346d7bf9ddd078 and e264de4a5ba146c1331a4f54d764d8cd15c007d4.

Before

```
function transferFrom(address from, address to, uint256 tokenId) public {
        require(false, "Only the contract can make transfers");
        from;
        to;
        tokenId;
}

function safeTransferFrom(address from, address to, uint256 tokenId, bytes memory _data) public {
        require(false, "Only the contract can make transfers");
        from;
        to;
        tokenId;
}
```

After

```
function transferFrom(address from, address to, uint256 tokenId) public checkState(States.WITHDRAW) onlyTokenOwner(tokenId) {
        _transferFrom(from, to, tokenId);
}

function safeTransferFrom(address from, address to, uint256 tokenId, bytes memory _data) public checkState(States.WITHDRAW)
onlyTokenOwner(tokenId) {
        _transferFrom(from, to, tokenId);
        _data;
}
```

The two functions are not used within the contract, as the contract is using the internal "_transferFrom()" function to transfer tokens during the other states. The contract is only in the "withdraw" state after "determineWinner()" has been called. All external/public functions are correctly checking whether the correct state is set, so that we can't see any way to abuse this. We **don't** think that this change has introduced any new risks or flaws.

Disclaimer

in this report is provided "as is" without warranty, representation, or guarantee of any kind, including the 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 →