byterocket

REALITY CARDS

RealityCards v2

September 18, 2021

1. Preface

The team of **RealityCards** contracted byterocket to conduct a smart contract audit of RealityCards V2. RealityCards is developing a novel prediction and betting market based on NFTs. In their latest update, they introduced various new features, such as an order book-based rent mechanism and a treasury that handles the financial part.

The team of byterocket reviewed and audited the above smart contracts in the course of this audit. We started on the 19th of August and finished on the 18 of September, 2021.

The audit included the following services:

- *Manual Multi-Pass Code Review*
- *Automated Code Review*
- *In-Depth Protocol Analysis*
- *Deploy & Test on our Testnet*
- *Formal Report*

byterocket gained access to the code via their public GitHub repository. We based the audit on the master branch's state on September 2nd, 2021 (*commit hash* *4d2316e4628425f35edae26f1e0627a4142d053b*).

2. Manual Code Review

We conducted a manual multi-pass code review of the smart contracts mentioned in section (1). Three different people went through the smart contract independently and compared their results in multiple concluding discussions.

These contracts are written according to the latest standards used within the Ethereum community and the Solidity community's best practices. The naming of variables is very logical and understandable, which results in the contract being useful to understand. The code is very well documented in the code itself.

A document containing an extensive description of the code and its specifications has been handed over to us prior to the start of our audit. This document has been beneficial in assessing the different functions of the implementation.

Any findings in prior versions of this document have been sufficiently addressed by the development team to our satisfaction. We have not been able to find any issues since.

Hence, on the code level, we **found no bugs or flaws**. A further check with multiple automated reviewing tools (*MythX*, *Slither*, *Manticore*, *and different fuzzing tools*) **did not find any additional bugs** besides a lot of common false positives.

2.1. Bugs & Vulnerabilities

2.1.A - RCMarket.sol - Line 637 [LOW SEVERITY] [FIXED]

```
function rentAllCards(uint256 _maxSumOfPrices) external
```

The frontrun protection in this function sums up the current prices of the tokens, which the user inputs as maxSumOfPrices. However, if a user expects that this is the final price that they will pay, this check will fail, as it doesn't take the 10% increase into account and subsequently fails. We would suggest renaming it to make this clear (so something like maxSumOfOldPrices) or to account for this in the calculation.

2.1.B - RCMarket.sol - Line 817 [HIGH SEVERITY] [FIXED]

```
function sponsor(address _sponsorAddress, uint256 _amount)
```

Given the fact that in line 808 there is already a sponsor function that can be called by users to sponsor and the comment in line 815 notes that this function here ought to be called by the factory only, we highly suggest adding this check in. Currently this would potentially allow anyone to call this with the address of other users, given that they have a standing allowance. It looks like this check was just forgotten.

2.1.C - RCTreasury.sol - Line 309 [LOW SEVERITY] [FIXED]

```
if ((user[_user].deposit + _amount) >
 (user[_user].bidRate / minRentalDayDivisor))
```

This foreclosure check in the deposit() function does not work properly, since the _amount has already been added to the deposit in line 303, hence rendering this check incorrect. We would advise you to remove the second adding of the _amount.

2.1.D - RCTreasury.sol - Line 309 [LOW SEVERITY] [FIXED]

```
function topupMarketBalance(uint256 _amount) external override
```

This function influences the balances of the contract(s) but does not make use of the balancedBooks modifier, which we would advise to do. The other functions that do influence the balances do this.

2.1.E - RCTreasury.sol - Line 703 [MEDIUM SEVERITY] [FIXED]

```
function collectRentUser(address _user, uint256 _timeToCollectTo)
```

The _timeToCollectTo value is not being checked, hence a malicious caller could set this to a very high value to foreclose competing users. This is probably not desired and the value should (at least we think) be limited to the current block.timestamp, so potentially users could call this but not do any harm. We would advise you to either remove the input and just take the current block.timestamp automatically or limit it accordingly.

2.2. Other Remarks and Findings

2.2.A - RCTreasury.sol - Line 600 [NO SEVERITY] [FIXED]

```
function resetUser(address _user) external override onlyOrderbook
```

This function changes the foreclosure flag of a user back to false, which might change the foreclosure state of a user. The corresponding event however is not emitted, which it probably should.

```
event LogUserForeclosed(address indexed user, bool indexed foreclosed);
```

3. Protocol/Logic Review

Part of our audits are also analyses of the protocol and its logic. A team of three auditors went through the implementation and documentation of the implemented protocol.

Treasury-based Approach

In contrast to the first version of the RealityCards smart contracts, all of the xDai related actions are now facilitated through a treasury contract. This not only reduces the attack surface but also allows for a way better UX. It shifts the security model of the whole system a bit towards a more central approach, where a potential attack would influence all of the markets simultaneously, which is (*in our opinion*) outweighed by the benefits that it provided. The development team has implemented various security measures to protect the contract, which we deem sufficient at this point.

Outside-Order-Book Approach

In an earlier implementation, the order book was part of the RCMarket contract itself. With the growing complexity of this contract and the overall complexity of the project, this part is now being located in a separate contract. This made it easier for us to separate the logic itself and test it. We haven't found any problems in the implementation, as it did change quite a bit but is still based on the same logic.

Cross-Chain Communication

As the RealityCards implementation migrated from the Ethereum Mainnet to the xDai chain earlier last year, the communication between these two networks is always somewhat of a specialty. We have seen many projects suffering from problems when cross-chain bridges are (*for some arbitrary reason*) failing or suffering from an outage. We generally advise all of our clients to assume that every bridge call will fail and keep this in mind when developing with bridges. The development team of RealityCards has done a great job in doing so - all of the essential functions can be called multiple times in case of bridge errors or outages. The receiving side will always accommodate numerous calls.

Meta-Transactions

The team of RealityCards followed a very widely-accepted approach in implementing their Meta Transaction mechanism,

which allows users to execute gasless transactions and interact with the platform without needing to have the xDai chain enabled in their wallet. We were not able to identify problems that this would cause.

General

We could **not find any weaknesses or underlying problems** of a game-theoretic nature in the protocol and its logic. We were also not able to think of any loopholes that the RealityCards developers did not cover.

4. Testnet Deployment/Logic Review

As per our testing strategy, we deploy audited smart contracts (*if requested by the client*) onto a testnet to verify the implementation's functionality. We usually deploy simple smart contracts to a local Ganache instance, which is sufficient for most cases. In this case, given the contracts' complexity, we wanted to ensure no Ganache-related coverups of any misbehavior of the contracts. We created two testnets: a geth-based one and a parity/openethereum-based one. All of our tests have been **executed on both testnets without any difference in the results**. We were able to use the contracts as intended and could not maliciously game the protocol in practice.

We used fuzzing tools that generate random and semi-random inputs and interact with the contracts, trying to produce any unforeseen states within the contracts, especially the treasury contract. Throughout our tests, we were **not able to create or observe any issues or problems**. The contract behaved as expected and reverted correctly, given wrong inputs.

5. Summary

During our code review (*which was done manually and automated*), we **found no bugs or flaws**. Additionally, our automated systems and review tools also **did not find any additional ones**.

The review and analysis of the protocol did neither uncover any game-theoretical nature problems nor any other functions prone to abuse.

During our multiple deployments to various local testnets we haven't been able to find any additional problems or unforeseen issues.

In general, we are **delighted** with the overall quality of the code and its documentation.

Disclaimer

As of the date of publication, the information provided in this report reflects the presently held understanding of the auditor's knowledge of security patterns as they relate to the client's contract(s), assuming that blockchain technologies, in particular, will continue to undergo frequent and ongoing development and therefore introduce unknown technical risks and flaws. The scope of the audit presented here is limited to the issues identified in the preliminary section and discussed in more detail in subsequent sections. The audit report does not address or provide opinions on any security aspects of the Solidity compiler, the tools used in the development of the contracts or the blockchain technologies themselves, or any issues not specifically addressed in this audit report.
The audit report makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, the legal framework for the business model, or any other statements about the suitability of the contracts for a particular purpose, or their bug-free status.

To the full extent permissible by applicable law, the auditors disclaim all warranties, express or implied. The information 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

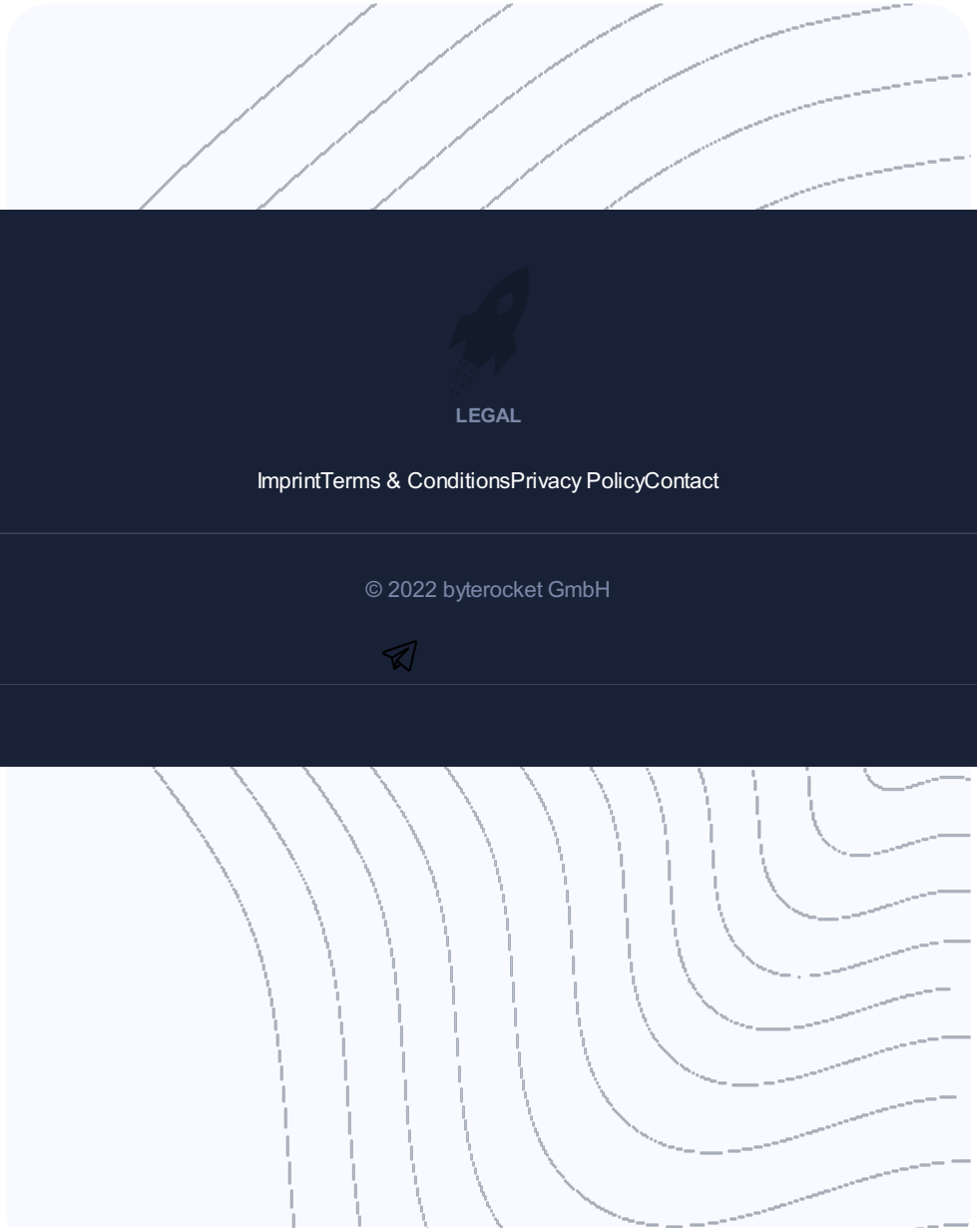
We store our public audit reports on IPFS; a peer-to-peer network called the "**I**nter **P**lanetary **F**ile **S**ystem". This allows us to store our reports in a distributed network instead of just a single server, so even if our website is down, every report is still available.

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 →



© 2022 byterocket GmbH