Reality Cards
Findings & Analysis Report

2021-08-02

Table of contents

Overview

About C4

Code 432n4 (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the code contest outlined in this document, C4 conducted an analysis of Reality Cards smart contract system written in Solidity. The code contest took place between June 9 and June 16, 2021.

Wardens

12 Wardens contributed reports to the Reality Cards code contest:

1. 0xRajeev
2. gpersoon
3. cmichel
4. a_delamo
5. shw
6. pauliax
7. maplesyrup (heiho1 and thisguy__)
8. heiho1
9. jvaqa
10. Jmukesh
11. s1m0
12. axic

This contest was judged by LSDan.

Final report assembled by ninek and moneylegobatman.

Summary

The C4 analysis yielded an aggregated total of 75 unique vulnerabilities. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity, 17 received a risk rating in the category of MEDIUM severity, and 18 received a risk rating in the category of LOW severity.

C4 analysis also identified 29 non-critical recommendations.

Scope

The code under review can be found within the C4 Reality Cards code contest repository is comprised of 24 smart contracts written in the Solidity programming language.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (4)

[H-01] Unchecked ERC20 transfers can cause lock up

Submitted by axic*, also found by* gpersoon*,* pauliax*, [Jmukesh] (*https://twitter.com/MukeshJeth*),* a_delamo*,* s1m0*,* cmichel*, and* shw_

Some major tokens went live before ERC20 was finalized, resulting in a discrepancy whether the transfer functions should (A) return a boolean or (B) revert/fail on error. The current best practice is that they should revert, but return "true" on success. However, not every token claiming ERC20-compatibility is doing this — some only return true/false; some revert, but do not return anything on success. This is a well known issue, heavily discussed since mid-2018.

Today many tools, including OpenZeppelin, offer a wrapper for "safe ERC20 transfer":

RealityCards is not using such a wrapper, but instead tries to ensure successful transfers via the `balancedBooks` modifier:

```
    modifier balancedBooks {
        _;
        // using >= not == in case anyone sends tokens direct to cor
        require(
            erc20.balanceOf(address(this)) >=
                totalDeposits + marketBalance + totalMarketPots,
            "Books are unbalanced!"
        );
    }
```

This modifier is present on most functions, but is missing on `topupMarketBalance`:

```
    function topupMarketBalance(uint256 _amount) external override {
        erc20.transferFrom(msgSender(), address(this), _amount);
        if (_amount > marketBalanceDiscrepancy) {
            marketBalanceDiscrepancy = 0;
        } else {
            marketBalanceDiscrepancy -= _amount;
        }
        marketBalance += _amount;
    }
```

In the case where an ERC20 token which is not reverting on failures is used, a malicious actor could call `topupMarketBalance` with a failing transfer, but also move the value of `marketBalance` above the actual holdings. After this, `deposit`, `withdrawDeposit`, `payRent`, `payout`, `sponsor`, etc. could be locked up and always failing with "Books are unbalanced".

Anyone can call `topupMarketBalance` with some unrealistically large number, so that `marketBalance` does not overflow, but is above the actually helping balances. This is only possible if the underlying ERC20 used is not reverting on failures, but is returning "false" instead.

Recommended Steps:

1. Use something like OpenZeppelin's `SafeERC20`
2. Set up an allow list for tokens, which are knowingly safe
3. Consider a different approach to the `balancedBooks` modifier

Splidge (Reality Cards) confirmed:

> The particular ERC20 contracts we are using don't have this issue. However for futureproofing in the event we change ERC20 tokens we will implement the recommended mitigation 1 and start using OpenZeppelin's SafeERC20.

Splidge (Reality Cards) resolved:

> Fix implemented here

[H-02] Can access cards of other markets

Submitted by gpersoon

Within `RCMarket.sol` the functions `ownerOf` and `onlyTokenOwner` do not check if the `_cardId/_token` is smaller than `numberOfCards`. So it's possible to supply a larger number and access cards of other markets. The most problematic seems to be `upgradeCard`. Here the check for `isMarketApproved` can be circumvented by trying to move the card via another market.

You can still only move cards you own.

```
338
339     function ownerOf(uint256 _cardId) public view override r
340         uint256 _tokenId = _cardId + totalNftMintCount; // d
341         return nfthub.ownerOf(_tokenId);
342     }
343
313 https:
314   modifier onlyTokenOwner(uint256 _token) {
315         require(msgSender() == ownerOf(_token), "Not owner")
316         _;
317     }
318
319 function upgradeCard(uint256 _card) external onlyTokenOwner(
320     _checkState(States.WITHDRAW);
321     require(
322         !factory.trapIfUnapproved() ||
323             factory.isMarketApproved(address(this)),    // th
324         "Upgrade blocked"
325     );
326     uint256 _tokenId = _card + totalNftMintCount;     // _car
327     _transferCard(ownerOf(_card), address(this), _card); //
328     nfthub.withdrawWithMetadata(_tokenId);
329     emit LogNftUpgraded(_card, _tokenId);
330 }
```

Recommend adding the following to `ownerOf`: `require(_card < numberOfCards, "Card does not exist");`

[Splidge (Reality Cards) confirmed but recommended *higher* severity]():

> I would assign this a higher severity level, I think it should be 3(High Risk) as this can be used to steal assets. An NFT being an asset as defined in the warden judging criteria found here.

> It is planned that eventually market creation will be opened up to anyone. There are several steps along this path towards opening up market creation:

1. only the Factory `owner` can create markets

2. Governors will be assigned who also have the ability to create markets
3. Anybody can be allowed to create markets by calling `changeMarketCreationGovernorsOnly`
4. NFTs allowed to be created (or more accurately not burned on market completion) by anyone by calling `changeTrapCardsIfUnapproved`

> The key here is that even in step 3 where anybody can create a market, the market will still require Governor approval for it to be displayed in the UI and for the NFT to be allowed to be upgraded. It is here in step 3 that `upgradeCard` could be called on an approved market in order to move a card from an unapproved market.

mcplums (Reality Cards) confirmed:

> Agreed, this indeed should have a higher severity- fantastic catch @gpersoon!!

Splidge (Reality Cards) resolved:

> Fixed here Impressed also with the simplicity of the solution.

dmvt (Judge) commented:

> Agree with the higher severity

[H-03] anyone can call function `sponsor`

Submitted by **paulius.eth**, *also found by* **0xRajeev**, **cmichel**, *and* **shw**

This function `sponsor` should only be called by the factory, however, it does not have any auth checks, so that means anyone can call it with an arbitrary `_sponsorAddress` address and transfer tokens from them if the allowance is > 0:

```
        /// @notice ability to add liqudity to the pot without being
        /// @dev called by Factory during market creation
        /// @param _sponsorAddress the msgSender of createMarket in
        function sponsor(address _sponsorAddress, uint256 _amount)
            external
            override
        {
```

```
            _sponsor(_sponsorAddress, _amount);
        }
```

Recommend checking that the sender is a factory contract.

Splidge (Reality Cards) confirmed:

> This is a good one!

mcplums (Reality Cards) commented:

> Yeah this is massive one!! Thanks @pauliax :)

Splidge (Reality Cards) resolved:

> fixed here

[H-04] Anyone can affect deposits of any user and turn the owner of the token

*Submitted by [a*dlamo](https://twitter.com/a_delamo)_

On `RCTreasury`, we have the method `collectRentUser`. This method is public, so anyone can call it using whatever user and whatever timestamp. So, calling this method using `user = XXXXX` and `_timeToCollectTo = type(uint256).max`, would make `isForeclosed[user] = true`.

See issue page for referenced code

Now, we can do the same for all the users bidding for a specific token. Finally, I can become the owner of the token by just calling `newRental` and using a small price. `newRental` will iterate over all the previous bid and will remove them because there are foreclosed.

Recommend that `collectRentUser` should be private and create a new public method with `onlyOrderbook` modifier.

Splidge (Reality Cards) confirmed:

> I like this. Although I might change the mitigation steps. I like keeping `collectRentUser` available to use, we can call it from our bot and it'll help keep user deposits updated in a timely manner for the frontend. I think I'll just add in

```
require(_timeToCollectTo <= block.timestamp, "Can't collect futı
```

mcplums (Reality Cards) commented:

> Yeah this is a real doozie, very happy this one was spotted!! Thanks @a_delamo :)

Splidge (Reality Cards) resolved:

> Fix implemented here

Medium Risk Findings

[M-01] payout doesn't fix `isForeclosed` state

Submitted by gpersoon

The function payout of `RCTreasury.sol` doesn't undo the `isForeclosed` state of a user. This would be possible because with a payout a user will receive funds so he can lose his `isForeclosed` status.

For example the function `refundUser` doesn't check and update the `isForeclosed` status in `RCTreasury` on L429 and line 447.

Recommend checking and updating the `isForeclosed` state in the payout function.

Splidge (Reality Cards) confirmed and suggested upgrading from 0 to 2 severity:

> The severity of this could be increased as a user might have believed that the payout would cancel their foreclosure. This could at a push count as 2 (Medium

> risk) because the “availability could be impacted” as in the definition here. This is because the user wouldn’t be allowed to place new bids without calling some other function that will cancel their foreclosure first.

dmvt (Judge) agreed with sponsor and upgraded from 0 to 2 severity:

Splidge (Reality Cards) resolved:

> Fixed here

[M-02] Critical `uberOwner` address changes should be a two-step process

Submitted by 0xRajeev, *also found by* gpersoon *and [a*dlamo] (https://twitter.com/a_delamo)._

As specified, `uberOwners` of `Factory`, `Orderbook` and `Treasury` have the highest privileges in the system because they can upgrade contracts of `market`, `Nfthub`, `order book`, `treasury`, `token` and `factory` which form the critical components of the protocol.

The contracts allow for `uberOwners` to be changed to a different address from the contract owner/deployer using the `changeUberOwner()` function which is callable by the current `uberOwner`. While this function checks for zero-address, there is no validation of the new address being correct. If the current `uberOwner` incorrectly uses an invalid address for which they do not have the private key, then the system gets locked because the `uberOwner` cannot be corrected and none of the other functions that require `uberOwner` caller can be executed.

Impact: The current `uberOwner` uses a non-zero but incorrect address as the new `uberOwner`. This gets set and now the system is locked and none of the `uberOwner`-only callable functions are callable. This error cannot be fixed either and will require redeployment of contracts which will mean that all existing markets have to be terminated. The system will have to be shut and restarted completely from scratch which will take a reputation hit and have a serious technical and business impact.

Recommend changing the single-step change of `uberOwner` address to a two-step process where the current `uberOwner` first approves a new address as a `pendingUberOwner`. That `pendingUberOwner` has to then claim the ownership in a separate transaction which cannot be done if they do not have the correct private key. An incorrectly set `pendingUberOwner` can be reset by changing it again to the correct one who can then successfully claim it in the second step.

Splidge (Reality Cards) marked as duplicate:

> Duplicate of #5

dmvt (Judge) commented:

> There is a very low probability coupled with a very high impact, making this a Medium risk issue in my opinion.

Note: *Additional conversation regarding this vulnerability can be found* here

[M-03] Missing `balancedBooks` modifier could result in failed system insolvency detection

Submitted by 0xRajeev, *also found by* gpersoon *and* paulius.eth

The `balancedBooks` modifier is used to "check that funds haven't gone missing during this function call" and is applied to `deposit`, `withdrawDeposit`, `payRent`, `payout` and `sponsor Treasury` functions which move funds in and out of the Treasury or adjust its market/user balances.

However, this modifier is missing in the `refundUser()` and `topupMarketBalance()` functions which also perform similar actions. The impact is that any miscalculations in these functions will lead to the system becoming insolvent.

Recommend adding modifier to the two functions above where it is missing.

Splidge (Reality Cards) confirmed and resolved in a duplicate issue:

> implemented here

Note: *Additional conversation regarding this vulnerability can be found here*

[M-04] `minRentalDayDivisor` can be different between markets and treasury

Submitted by gpersoon, also found by maplesyrup (heiho1 and thisguy__) and paulius.eth

The `minRentalDayDivisor` is defined in `RCTreasury.sol` and copied to each market. The `minRentalDayDivisor` can be updated via `setMinRental`, but then it isn't updated in the already created market.

To calculate the minimum rent time, in function `withdrawDeposit` of `RCTreasury.sol`, the latest version of `minRentalDayDivisor` is used, which could be different than the values in the market. So the markets will calculate the minimum rent time different. This could lead to unexpected results

```
    function initialize(
        ...
           minRentalDayDivisor = treasury.minRentalDayDivisor()
```

```
322 https:
323  function withdrawDeposit(uint256 _amount, bool _localWithdr
324 ...
325   require( user[_msgSender].bidRate == 0 || block.timestamp
326 ..
327  if ( user[_msgSender].bidRate != 0 &&  user[_msgSender].bid
328 ..
329
169
170   function setMinRental(uint256 _newDivisor) public override
171         minRentalDayDivisor = _newDivisor;
172     }
```

Recommend either accepting or at least documenting the risk of change to code to prevent this from happening.

Splidge (Reality Cards) acknowledged:

> Yes, This became apparent recently when we changed the `minRentalDayDivisor` during a beta test. Ideally this value is never changed and if it is changed then it will be done very infrequently. The main protection `minRentalDayDivisor` offers is against a DoS attack whereby an attacker gains some ownership time on a card and then will fill the orderbook with bids using sybil accounts (withdrawing almost all deposit after placing the bids), without `minRentalDayDivisor` these low value (but legitimate) bids would prevent other users from gaining ownership of the card (due to gas limits there's a limit to the rental collections we can perform) and give the attacker a greater share of the prize pot. The benefit of `minRentalDayDivisor` is that now these are zero value bids which are eligible for immediate deletion, and so there is now more of a cost to the attack which scales with the cost of the rental prices (which will closely be linked to the value of the prize pot). To this end `minRentalDayDivisor` is at it's most useful in the Treasury where it's main purpose is fulfilled in `withdrawDeposit()`, the usage in the markets is less beneficial and wasn't considered worth the extra gas usage to have the Markets fetch the updated value given the infrequency we will be changing it. We have accepted this risk.

dmvt (Judge) upgraded severity from 1 to 2:

> Updating to Medium risk to match the other reporting wardens: "Possible accidental loss of funds or information due to code manipulation or bad side effects of not properly outlining a payable function"

[M-05] `RCFactory.createMarket()` does not enforce `_timestamps` and `_timestamps` being larger than `_timestamps`, even though proper functioning requires them to be so

Submitted by jvaqa, *also found by* 0xRajeev, paulius.eth *and* shw

`RCFactory.createMarket()` does not enforce `_timestamps` [1] and `_timestamps` [2] being larger than `_timestamps` [0], even though proper functioning requires them to be so.

`IRCMarket` defines a sequence of events that each market should progress through sequentially, CLOSED, OPEN, LOCKED, WITHDRAW. (1)

The comments explicitly state that `_incrementState()` should be called "thrice" (2)

However, it is possible to create a market where these events do not occur sequentially.

You can create a market where the `marketOpeningTime` is later than the `marketLockingTime` and `oracleResolutionTime`.

This is because although `RCFactory` checks to ensure that `_timestamps[2]` is greater than `_timestamps[1]`, it does not check to ensure that `_timestamps[1]` is greater than `_timestamps[0]` (3)

This is also because although RCFactory checks to ensure that `_timestamps[0]` is equal to or greater than `block.timestamp`, it makes no check for a minimum value for `_timestamps[1]` or `_timestamps[2]`, or a relative check between the value of `_timestamps[0]` and `_timestamps[1]`. (4)

Thus, you can create a market where the `marketLockingTime` and the `oracleResolutionTime` occur before the `marketOpeningTime`.

When calling `RCFactory.createMarket()`, Alice can supply 0 as the argument for `_timestamps[1]` and `_timestamps[2]`, and any value equal to or greater than `block.timestamp` for `_timestamps[0]` (5)

Recommend adding the following check to `RCFactory.createMarket()`:

```
require(
    _timestamps[0] < _timestamps[1],
    "market must begin before market can lock"
);
```

Splidge (Reality Cards) confirmed and resolved:

> Implemented here

[M-06] Possible locked-ether (funds) Issue in `RCOrderbook.sol`

Submitted by maplesyrup (heiho1 and thisguy__)

When running the analyzer code, the following functions were found in `RCOrderbook.sol` to possibly lock funds due to it being a payable function with no withdraw function associated. See Issue #43 for more details.

Splidge (Reality Cards) confirmed initially and then disputed:

> I initially confirmed this because we aren't using the native currency on Matic/Polygon. However I think this should be disputed mainly because this function is used to call other functions which might be payable, although I admit currently we don't have payable functions, we might add them in the future. This library is used across all our contracts, had we put a payable function in the Treasury for instance, would this be considered a flaw to have this same library imported into the Orderbook?

Splidge (Reality Cards) commented:

> Note that the duplicate issue #51 was submitted by the same user.

dmvt (Judge) commented:

> Agree with the sponsor's explanation, but the issue exists regardless. Adding a way to retrieve locked funds would mitigate the issue.

[M-07] `maxSumOfPrices` check is broken

Submitted by OxRajeev

`rentAllCards()` requires the sender to specify a `_maxSumOfPrices` parameter which specifies "limit to the sum of the bids to place" as specified in the Natspec @param comment. This is apparently for front-run protection.

However, this function parameter constraint for `_maxSumOfPrices` is broken in the function implementation which leads to the total number of bids placed greater than the `_maxSumOfPrices` specified.

The impact of this is that the user may not have sufficient deposited, be foreclosed upon and/or impacted on other bids/markets.

Scenario: Assume two cards for a market with current winning rentals of 50 each. `_maxSumofPrices` = 101 passes check on L643 but then the forced 10% increase on L650 (assuming sender is not the owner of either card) causes `newRentals` to be called with 55 for each card thus totalling to 110 which is > 101 as requested by the user.

Recommend modifing the max sum of prices check logic to consider the 10% increase scenarios. Document and suggest the max sum of prices for the user in the UI based on the card prices and 10% requirement depending on card ownership.

Splidge (Reality Cards) confirmed and resolved:

> fixed **here**

[M-08] Flows can bypass market and global pause

Submitted by **OxRajeev**

Ability to pause all token transfers and all state changes for contracts is a "guarded-launch" best-practice for emergency situations for newly launched projects. The project implements this using a `marketsPaused` flag per market and a `globalPause` flag across all markets.

While these prevent renting of cards in a specific market and deposit/withdraw/rent cards across all markets, there are still public/external functions that are unguarded by these flags which can affect contract state in paused scenarios that will make it hard/impossible to recover correctly from the emergency pause.

Examples include `topupMarketBalance()` and `refundUser()` in `Treasury` can be triggered even in a `globalPause` scenario. There could be other function flows

where it is not obvious that market/global pausing is enabled because it is enforced in one of the functions called deep within the code within one of the conditionals.

The impact is that markets get paused but the contracts cannot be restarted because of state changes affected during the pause via unguarded external/public functions.

Recommend applying `marketPaused` and `globalPause` check clearly in the beginning of all public/external functions which move tokens/funds in/out or change contract state in any way. Also, Validate all possible control flows to check that market/global pausing works in all scenarios and applies to all contract state and not specific functionalities.

Splidge (Reality Cards) disputed and disagreed with severity:

> `marketPause` is **declared** as only limiting rentals in a specific market.
> `globalPause` is **declared** as stopping deposits, withdraws and rentals across all markets. Therefore they are functioning as intended.

> Also, the example of `refundUser()` is not true, it will fail in a `globalPause` because it is only called by markets during a rent collection and a rent collection requires the calling of `payout` which is restricted by `globalPause`.

dmvt (Judge) commented:

> `topupMarketBalance` does appear to be a deposit of sorts. I think the warden's take on the issue is valid and sponsor should seriously consider looking closer at the potential side effects of not fully pausing intentional transfer functions.

[M-09] Deposit whitelist enforced on `msg.sender` instead of user

Submitted by **0xRajeev**

The Treasury `deposit()` function credits amount to the user address in parameter instead of the `msgSender()` function that is actually making the deposit with the

rationale (as explained in the Natspec comment) being that this may be called via contract or L1-L2 bot.

However, the deposit whitelist should ideally be enforced on the `_user` address. If `msgSender()` is blacklisted, user address can still `deposit()` from another whitelisted `msgSender()` address while retaining the user address that is used for leader boards and NFTs.

The impact of this is that even if the user misbehaves in interactions with the system (e.g. trolls, spams) and their corresponding `msgSender()` is removed from the whitelist. The user can continue to deposit into the system via another whitelisted `msgSender()` without any impact to leader boards or NFTs.

Recommend using whitelist on user address instead of `msgSender()`.

Splidge (Reality Cards) disputed and disagreed with severity:

> It is stated that the whitelist will "only allow certain addresses to deposit" here and that toggleWhitelist() allows an address to deposit here.

> I think that the whitelist is performing as intended, but thanks for this issue report as this could easily have been a larger issue.

> We only plan to use the whitelist as a very rudimentary barrier just for the initial launch. I think that only allowing certain addresses to deposit is sufficient for now. Maybe if time allows I'll make the changes but changing the whitelist to allow the `_user` instead of the `msgSender()` would also block contracts and layer1->layer2 bot, so there'd need to be exceptions made for them. I'd rather not play about with sensitive functions at the last minute when we aren't going to be using the whitelist much anyway.

dmvt (Judge) commented:

> Warden makes a good point. This could allow griefing of other parts of the system. If the barrier winds up being needed longer than expected or users act in unexpected ways, sponsor may wind up wishing they had reconsidered addressing this. Obviously, sponsor is free to ignore, but the issue seems to be a valid one with significant potential impact.

[M-10] Missing call to `removeOldBids` may affect foreclosure

Submitted by OxRajeev

`Orderbook.removeBids()` as commented:

```
    ///remove bids in closed markets for a given user
    ///this can reduce the users `bidRate` and chance to foreclose
```

`removeOldBids()` is performed currently in `Market.newRental()` and `Treasury.deposit()` to "do some cleaning up, it might help cancel their foreclosure" as commented. However, this is missing in the `withdrawDeposit()` function where the need is the most because user is removing deposit which may lead to foreclosure and is even commented as being useful on L356.

The impact is that, if we do not remove closed market bids during withdrawDeposit, the closed market bids still get accounted in user's `bidRate` in the conditional on L357 and therefore do not prevent the foreclosure in `withdrawDeposit` that may happen in L357-L367. User may get foreclosed because of mis-accounted closed-market bids in the order book.

Recommend adding call to `removeOldBids()` on L355 of `withdrawDeposit()` of Treasury.

Splidge (Reality Cards) confirmed but disagreed with severity and then resolved:

> This was intentionally left out in an older version of the contracts because of the way `withdrawDeposit` worked before we had the per-user rent collection. Added it back in again here.

[M-11] NFT Hub implementation deviates from ERC721 for transfer functions

Submitted by OxRajeev

ERC721 standard and implementation allows the use of approved addresses to affect transfers besides the token owners. However, the L2 NFT Hub implementation deviates from ERC721 by ignoring the presence of any approvers in the overriding function implementations of `transferFrom()` and `safeTransferFrom()`.

The impact is that the system interactions with NFT platforms may not work if they expect ERC721 adherence. Users who interact via approved addresses will see their transfers failing for their approved addresses.

Given that the key value proposition of this project is the use of NFTs, the expectation will be that it is fully compatible with ERC721.

Recommend adding support for approval in NFT transfers.

mcplums (Reality Cards) commented:

> This is a nice one, I see no reason why we can't implement this

Splidge (Reality Cards) confirmed:

> Yes, we will need to add this, although we will need to override the approvals until the market has locked and the cards true owner is discovered.

Splidge (Reality Cards) resolved:

> I've changed from overriding specific functions which could be dangerous if we were to upgrade to an OpenZeppelin implementation that had alternative transfer functions. Now we use the `_beforeTokenTransfer` hook and check that only the factory or the market can do a transfer before the market has entered the withdraw state. Implemented here

[M-12] `RCNftHubL2.safeTransferFrom` not according to spec

Submitted by **cmichel**, *also found by* **0xRajeev**

The `RCNftHubL2.safeTransferFrom` function does not correctly implement the ERC721 spec:

> When using `safeTransferFrom`, the token contract checks to see that the receiver is an IERC721Receiver, which implies that it knows how to handle ERC721 tokens. ERC721

This check is not implemented, it just drops the `_data` argument.

Contracts that don't know how to handle ERC721 tokens (are not an `IERC721Receiver`) can accept them but they should not when using `safeTransferFrom` according to spec.

Recommend Implementing the `IERC721Receiver` check in `safeTransferFrom`.

Splidge (Reality Cards) confirmed and resolved:

> This has been fixed while working on issue #118 commit here

[M-13] Wrong calculation on `_collectRentAction`

*Submitted by [a*dlamo](https://twitter.com/a_delamo)_

The method `_collectRentAction` contains the following code:

in case 6, it is doing:

```
_refundTime = block.timestamp - marketLockingTime;
```

instead of:

```
_refundTime = _timeUserForeclosed - marketLockingTime;
```

This could lead to funds being drained by the miscalculation.

mcplums (Reality Cards) commented:

> This is a really great find!!

Splidge (Reality Cards) confirmed and resolved:

> Fix implemented here

[M-14] Market-specific pause is not checked for sponsor

Submitted by cmichel

The treasury only checks its `globalPause` field but does not check its market-specific `marketPaused` field for `Treasury.sponsor`. A paused market contract can therefore still deposit as a sponsor using `Market.sponsor` and result in the market-specific pause not work correctly.

Recommend adding checks for `marketPaused` in the Treasury for `sponsor`.

mcplums (Reality Cards) commented:

> I don't think this is important but I guess it can't hurt to block sponsorship if paused

Splidge (Reality Cards) confirmed but disagreed with severity:

> I'm not sure why this is a severity 2? Maybe it should be lower. Sponsoring a market, whether paused or not, doesn't come with an expectation to receive the funds back. So assets are not at risk here.

Splidge (Reality Cards) resolved:

> There have been changes made to `marketPaused` and how markets are created due to other issues that have been found. By default markets are now created in a paused state and it'd be useful to be able to sponsor them before the governors approve them. It's a nice thing for the sponsorship to be in place before anybody interacts with the contract. I have however made changes such that is the market pause is ever turned on by the Treasury owner then the sponsor function will revert. Changes here

[M-15] Deposits don't work with fee-on transfer tokens

Submitted by cmichel

There are ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`.

The `deposit()` function will introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

Recommend measuring the asset change right before and after the asset-transferring routines as a possible mitigation.

mcplums (Reality Cards) commented:

> I think balancedBooks modifier should handle this?

> Of course it means we are unable to use such tokens, but that is ok

****Splidge (Reality Cards) disputed**

> oh, trying the same one again..? 😁 https://github.com/code-423n4/2021-05-88mph-findings/issues/16

> I'll fight this one though, I'd argue that we are using ERC20 tokens and according to the ERC20 spec for transferFrom:

> Transfers `_value` amount of tokens from address `_from` to address `_to`

> A deflationary token therefore isn't compliant to ERC20 as it doesn't transfer the full `_value` and so it isn't what we are planning to use and not relevant here.

dmvt (Judge) commented:

> If you plan not to support these tokens it should be very clearly documented. Keep in mind that "we don't support that" still has massive impact on the users involved. See: imBTC / ERC777 on Uniswap v1. The issue is valid and should stand in the audit report, in part so that future users see it.

[M-16] Deposits can be denied by abusing `maxContractBalance`

Submitted by cmichel

The treasury implements a max contract balance check in the `deposit` function:

```
    require(
        (erc20.balanceOf(address(this)) + _amount) <= maxContractBal
        "Limit hit"
    );
```

A whale can stop anyone from depositing by front-running a user's deposit with a deposit that pushes the contract balance to the `maxContractBalance` limit first. The user's deposit will then fail in the check. Afterwards, the whale can withdraw again.

> This is not only restricted to whales, miners/users can do the same using same-block cross-transaction flashloans and submitting a `(attacker deposit, user deposit, attacker withdraw)` flashbundle to a miner. Possibilities like this will only become more prevalent in the future.

Any users can be blocked from depositing which prevents them from renting cards. This allows an attacker to manipulate the outcome of a market in their favor by strategically preventing other competitors to bid on their cards (causing forfeiture due to a low deposit balance).

Recommend removing the contract limit or at least set the limit very high if it keeps happening.

mcplums (Reality Cards) acknowledged:

> This is a good one- but I don't think we need to make any changes to the contract. We can use it as originally intended, then if it is exploited as above, we can switch to only setting the variable to 0 or maxuint256. So it just acts as a toggle on whether deposits are allowed.

[M-17] Function `foreclosureTimeUser` returns a shorter user's foreclosure time than expected

Submitted by shw

The function `foreclosureTimeUser` of `RCTreasury` underestimates the user's foreclosure time if the current time is not the user's last rent calculation time. The underestimation of the foreclosure time could cause wrong results when determining the new owner of the card.

The variable `timeLeftOfDeposit` at line 668 is calculated based on `depositAbleToWithdraw(_user)`, the user's deposit minus the rent from the last rent calculation to the current time. Thus, the variable `timeLeftOfDeposit` indicates the time left of deposit, starting from now. However, at line 672, the `foreclosureTimeWithoutNewCard` is calculated by `timeLeftOfDeposit` plus the user's last rent calculation time instead of the current time. As a result, the user's foreclosure time is reduced. From another perspective, the rent between the last rent calculation time and the current time is counted twice.

Recommend changing `depositAbleToWithdraw(_user)` at line 669 to `user[_user].deposit`. Or, change `user[_user].lastRentCalc` at both line 672 and 678 to `block.timestamp`.

Splidge (Reality Cards) confirmed and resolved:

> phew, this was one to wrap your head around. I went with the first recommended mitigation because I believe the second one could causes issues if the user had already foreclosed, `depositAbleToWithdraw` would return 0 and so `foreclosureTimeWithoutNewCard` would incorrectly show as `block.timestamp`. Fix implemented here

> Really nice spot this one. Many thanks for such an in-depth look into the maths.

Low Risk Findings

[L-01] Use of `assert()` instead of `require()`

Submitted by OxRajeev, *also found by* jvaqa, *[Jmukesh]* (https://twitter.com/MukeshJeth) and cmichel_

Contracts use `assert()` instead of `require()` in multiple places. This causes a Panic error on failure and prevents the use of error strings.

Prior to solc 0.8.0, `assert()` used the invalid opcode which used up all the remaining gas while `require()` used the revert opcode which refunded the gas and therefore the importance of using `require()` instead of `assert()` was greater. However, after 0.8.0, `assert()` uses revert opcode just like `require()` but creates a `Panic(uint256)` error instead of `Error(string)` created by `require()`. Nevertheless, Solidity's documentation says:

> "Assert should only be used to test for internal errors, and to check invariants. Properly functioning code should never create a Panic, not even on invalid external input. If this happens, then there is a bug in your contract which you should fix. Language analysis tools can evaluate your contract to identify the conditions and function calls which will cause a Panic."

whereas

> "The require function either creates an error without any data or an error of type Error(string). It should be used to ensure valid conditions that cannot be detected until execution time. This includes conditions on inputs or return values from calls to external contracts."

Recommend using `require()` with informative error strings instead of `assert()`.

Splidge (Reality Cards) labeled duplicate (removed by judge):

Splidge (Reality Cards) confirmed and resolved in a separate issue:

> Most of these were fixed here Some extras found in issue #83 have been fixed here Although a number were left as they are correctly using assert()

Note: *Additional conversation regarding this vulnerability can be found* here

[L-02] Lack of zero address validation

Submitted by [Jmukesh](https://twitter.com/MukeshJeth), also found by 0xRajeev, maplesyrup (heiho1 and thisguy__), and cmichel_

The constructor of `RCorderbook.sol` lacks zero address validation, since parameter of constructor are used initialize state variable which are used in other functions of the contract, error in these state variable can lead to redeployment of contract.

Recommend adding a required condition to check for zero address.

Splidge (Reality Cards) confirmed:

> I think the zero address validation isn't a problem for `factoryAddress` as this can be set later in the function `setFactoryAddress` However yes Treasury is missing a possible `setTreasuryAddress`

Splidge (Reality Cards) resolved:

> implemented here Additional changes for #142 and #115 are here

[L-03] Multiple calls necessary for `getWinnerFromOracle`

Submitted by gpersoon*, also found by* 0xRajeev

Sometimes multiple calls necessary to `getWinnerFromOracle` are necessary to get the `_winningOutcome` to be processed:

- `getWinnerFromOracle` calls `setWinner`
- `setWinner` calls `lockMarket`
- `lockMarket` calls `collectRentAllCards`
- `collectRentAllCards` can return false, which means is has to be called again. In that case the `_winningOutcome` isn't processed and `getWinnerFromOracle` has to be called again.

It's not easy to determine how many times `getWinnerFromOracle` has to be called. (it can be seen via `emit LogWinnerKnown(winningOutcome)` , however this cannot be read from a smart contract)

Recommend letting the function `getWinnerFromOracle` return a boolean to indicate it has to be called again.

Splidge (Reality Cards) acknowledged:

> I have marked this as a duplicate of #4 because the core of the problem in both issues is that the rent collection might not have succeeded (completed) which would cause whatever the triggering function to silently fail. I believe that calling this from a smart contract wouldn't necessarily be a problem, partly because if the rent collection doesn't complete then you've used up most of the gas in the block anyway and wouldn't be able to call it again until the next block, but also there are public variables you could check to determine the success if performing this from a bot (as we intend to do for `getWinnerFromOracle`). Our frontend does use the events to determine the success of transactions (such as `updateTimeHeldLimit`) so this isn't an issue for us.

> For the time being I'll mark this as acknowledged and if time allows before launch I'll make the changes and come back to confirm the fix.

[L-04] `addToWhitelist` doesn't check `factoryAddress`

Submitted by **gpersoon**

The function `addToWhitelist` of `RCTreasury.sol` does a call to the `factory` contract, however the `factoryAddress` might not be initialized, because it is set via a different function (`setFactoryAddress`). The function `addToWhitelist` will revert when it calls a 0 address, but it might be more difficult to troubleshoot.

L233

```
    function setFactoryAddress(address _newFactory) external over
        ...
        factoryAddress = _newFactory;
```

```
}
```

L210

```
function addToWhitelist(address _user) public override {
    IRCFactory factory = IRCFactory(factoryAddress);
    require(factory.isGovernor(msgSender()), "Not authorisec
    isAllowed[_user] = !isAllowed[_user];
}
```

Recommend verifying that `factoryAddress` is set in the function `addToWhitelist`, for example using the following code. `require(factory != address(0), "Must have an address");`

Splidge (Reality Cards) confirmed and resolved:

> implemented here

[L-05] Deposit double-counting miscalculation could incorrectly prevent user foreclosure

Submitted by **0xRajeev**, *also found by* **gpersoon**, **paulius.eth** *and* **shw**

In the `deposit` function, the deposit `_amount` has already been added to the user's deposit on L303. The addition of `_amount` again to the deposit on L309 for checking against daily `bidRate` effectively leads to double counting of deposited `_amount` and may keep/bring user out of foreclosure even though they are not.

Scenario: Alice's current daily `bidRate` is 500 and deposit is 350. She makes a new deposit of 100 which should not bring her out of foreclosure because the new effective deposit will be 300+150 = 450 which is still less than 500. However, because of the double-counting miscalculation, the check performed is 450+100 > 500 which will pass and Alice is not foreclosed. She effectively gains double the deposit amount in treatment of deposits against foreclosure.

Recommend changing the conditional predicate on L309-310 from: `user[_user].deposit + _amount > user[_user].bidRate / minRentalDayDivisor` to: `user[_user].deposit > user[_user].bidRate / minRentalDayDivisor`

Splidge (Reality Cards) disagreed with severity:

dmvt (Judge) lowered risk from 2 to 1:

> I generally think the impact of this on the rest of the system is minimal. It results in a slight advantage to the user in foreclosure, but does not cause a loss or granting of additional funds. To take advantage of this exploit, the user would also need to be highly skilled at reading source code to find the exploit in the first place. Even if they took the time to do this, the effect would not be permanent. I'm aligned with the view expressed in #26 and #37 that this is low severity.

Splidge (Reality Cards) confirmed and resolved in a separate issue:

> fixed here

Note: *Additional conversation regarding this vulnerability can be found here*

[L-06] unnecessary emit of `LogUserForeclosed`

Submitted by **gpersoon**

The function deposit of `RCTreasury.sol` resets the `isForeclosed` state and emits `LogUserForeclosed`, if the use have enough funds. However this also happens if the user is not Foreclosed and so the emit is redundant and confusing.

L279

```
    function deposit(uint256 _amount, address _user) public override
        ....
            // this deposit could cancel the users foreclosure
            if ( (user[_user].deposit + _amount) > (user[_user].bidR
                isForeclosed[_user] = false;
                emit LogUserForeclosed(_user, false);
```

```
            }
            return true;
        }
```

Recommend only do the emit when `isForeclosed` was true

Splidge (Reality Cards) confirmed and resolved:

> implemented here

[L-07] Use of `ecrecover` is susceptible to signature malleability

Submitted by OxRajeev*, also found by [a*dlamo](https://twitter.com/a_delamo)_

The `ecrecover` function is used to verify and execute Meta transactions. The built-in EVM precompile `ecrecover` is susceptible to signature malleability (because of non-unique s and v values) which could lead to replay attacks

While this is not exploitable for replay attacks in the current implementation because of the use of nonces, this may become a vulnerability if used elsewhere.

Recommend considering using OpenZeppelin's ECDSA library (which prevents this malleability) instead of the built-in function:

Splidge (Reality Cards) confirmed and acknowledged:

> This issue will now be fixed, not because of the reasons discovered in this issue, but because of the reasons explained in #166

Splidge (Reality Cards) commented:

> Nevermind, issue #166 couldn't be solved using the recommended mitigation so this issue will remain unresolved for now.

[L-08] NFT minting limit dependence on block gas limit

Submitted by OxRajeev

The current block gas limit is 15M and not 12.5 as indicated in the comment for `setNFTMintingLimit(60)` in Factory's constructor. So this could be changed accordingly but a safe threshold needs to be enforced in the setter `setNFTMintingLimit()` which is currently lacking. That would prevent accidentally setting the minting limit to something beyond what the block gas limit would safely allow.

Recommend that If NFT minting limit dependence on block gas limit is critical to the functioning, consider using `GASLIMIT` opcode to dynamically check block gas limit to set `nftMintingLimit` appropriately before creating a market.

Splidge (Reality Cards) acknowledged:

> The minting limit isn't critical to the functioning, 60 is already far beyond what we anticipate is required. The contracts will actually be deployed on Matic/Polygon which has an even higher gas limit still (although adjustable).

[L-09] Basis points usage deviates from general definition

Submitted by **OxRajeev**

The general definition of basis points is 100 bps = 1%. The usage here, 1000 bps = 100%, deviates from generally accepted definition and could cause confusion among users/creators/affiliates or potential miscalculations.

Recommend documenting the used definition of basis points or switch to the generally accepted definition.

Splidge (Reality Cards) confirmed:

> Yep, I discovered this also looking at one of the other issues. These have been changed to PER_MILLE which is equivalent to a MegaBip

Splidge (Reality Cards) resolved:

> Corrected alongside #10 in the commit **here**

[L-10] Missing input validation on timeout

Submitted by OxRajeev

Factory constructor sets timeout to 86400 seconds but setter `setTimeout()` has no threshold checks for a min timeout value. If this is accidentally set to 0 or lower-than-safe value then there is no dispute window and users lose confidence in market.

Recommend adding input validation for threshold checks on both low and high ends.

Splidge (Reality Cards) acknowledged:

> The oracle doesn't allow a timeout of 0 so accidentally setting to 0 wouldn't cause market problems, a lower than safe value provides a warning to the users on the oracle. It is important that the users check the oracle anyway to get the specific wording of the question. In practice the owner and `uberOwner` will be a multisig and so a mistake would require multiple people to perform the same mistake.

[L-11] `isGovernor` excludes `Factory` owner

Submitted by OxRajeev

In the Factory contract, the `Factory` owner is authorized to change approval for governor addresses and is also treated as a governor in the modifier `onlyGovernors`. However, `isGovernor` modifier excludes `Factory` owner as a governor for some reason. This function is used only by `Treasury` to whitelist users who can deposit tokens and would make sense to include `Factory` owner as a governor to be consistent.

Without doing so, `Factory` owner cannot whitelist users without adding itself or someone else as a governor.

Recommend including `Factory` owner in `isGovernor()`.

Splidge (Reality Cards) acknowledged:

> While the owner has the ability to perform the tasks of a governor and the ability to make themselves a governor they might not actually be a governor so probably shouldn't be included in `isGovernor` as this could be used for other external reasons. The addition of the owner into `onlyGovernors` is more of a convenience for the owner as they have the power to change this anyway, why make it harder for them?

dmvt (Judge) commented:

> I think this issue is a fair critique. The language should probably be cleaned up so future devs don't get confused as to when 'governors' include the factory owner and when not.

[L-12] Making `isMarketApproved` False on an operational market will lock NFTs to L2

Submitted by **OxRajeev**

Once market is approved and operational, changing approval to false should not be allowed or else it will prevent NFTs from being withdrawn to mainnet. All other Governor controlled variables are used during market creation and not thereafter, except this one. The other `onlyGovernors` functions only affect state before market creation but this one affects after creation.

Recommend that once market is approved and operational, changing approval to false should not be allowed.

Splidge (Reality Cards) confirmed and resolved:

> The concept of trapping the NFTs has been removed and markets will now be created in the paused state. `marketPause` has been redone such that governors may only remove `marketPause` provided that it hasn't been set by the owner. This means once a governor approves a market it will be shown in the UI and unpaused, they may un-approve the market again however this will only hide it in the UI and does not pause it again. Implemented **here**

[L-13] Susceptible to collusion and sybil attacks

Submitted by OxRajeev

Collusion and sybil attacks are general problems with blockchain-based prediction markets and voting systems.

Collusion between market creator and bidders, where the creator creates a niche prediction market for which only they know the outcome with a higher degree of probability (than others) and either spawn fake users (sybils) to increase the pot size and lure victims to add bids. Creator or its fake users maintain the longest duration on the winning outcome (which they know with greater certainty than others) thus winning that market's outcome and taking the victim's rents (winner-take-all-mode).

The general problem is hard to solve. Recommend documenting and warning users suitably about risks involved.

Splidge (Reality Cards) disputed:

> This doesn't appear to be a problem with the code. There are warnings on the frontend. There is some quality control in the way markets are created by allowing governors to approve them and the question specifics must be clearly stated for the oracle. We have already had in a beta test the users disagree with the wording of an outcome and collectively invalidate the oracle thereby returning all rent paid to the users.

dmvt (Judge) commented:

> I'm going to let this one stand mostly to serve as an additional warning to users who take the time to read the audit report. I don't think there is any action to take beyond warnings on the frontend.

[L-14] Misplaced zero-address check

Submitted by OxRajeev

Misplaced zero-address check for `nfthub` on L595 in `createMarket()` because `nfthub` cannot be 0 at this point as `nfthub.addMarket()` on L570 would have already reverted if that were the case.

Recommend moving `nfthub` zero-address check to before the call to `nfthub.addMarket()`.

Splidge (Reality Cards) confirmed and resolved:

> fixed here

[L-15] Missing market open check

Submitted by OxRajeev

`Missing _checkState(States.OPEN)` on first line of `rentAllCards()` as specified on L617. These core market functions are supposed to operate only when market is open but the missing check allows control to proceed further in the control flow. In this case, the function proceeds to call `newRental()` which has a conditional check state == `States.OPEN` and silently returns success otherwise, without reverting.

The impact of this is that `rentAllCards` does not fail if executed when market is closed or locked. `newRental` returns silently without failure when market is closed or locked.

Add a `require()` to check market open state in the beginning of all core market functions and revert with an informative error string otherwise.

Splidge (Reality Cards) confirmed and resolved:

> I've added the check on `rentAllCards()` here. I have not made `newRental()` revert because we use this to lock the market if the market is beyond it's locking time. If the market does get locked successfully then the UI will update to show this. If it doesn't get locked then the appropriate accounting hasn't been completed yet.

[L-16] Assert indicates unnecessary check or missing constraint/logic

Submitted by OxRajeev

`updateLastRentalTime()` function “tracks when the user last rented so they cannot rent and immediately withdraw thus bypassing minimum rental duration.”

This function currently always returns true and so there is no need to assert its return value, as done in `newRental()`, unless it was meant to return false in some scenarios which indicates missing constraint/logic. It is not clear what that might be.

Given that the minimum rental duration is one of the two key protection mechanisms, any missing logic/constraint here could affect the project significantly.

Recommend validating constraint/logic to see if function should return false in any scenario. Remove assert at call site if otherwise.

Splidge (Reality Cards) comment:

> Duplicate of #55

dmvt (Judge) commented:

> I do not see how this is a duplicate of #55.

Splidge (Reality Cards) commented:

> Sorry, I think it must have been #53 I wanted to mark it a duplicate of. Although there is also some overlap with #83 as the assert wasn’t used correctly.

[L-17] `exitedTimestamp` set prematurely

Submitted by OxRajeev

The `exitedTimestamp` flag is used to prevent front-running of user exiting and re-entering in the same block. The setting of this flag in `exit()` should really be inside the conditionals and triggered only if current owner or if `bidExists`. It currently assumes that either of the two will always be true which may not necessarily be the case.

The impact of this is that a user accidentally exiting a card he doesn’t own or have a bid for currently, will be marked as exited and prevented from a `newRental` in the

same block. User can prevent one's own `newRental` from succeeding, because it was accidentally triggered, by front-running it himself with an exit. There could be other more realistic scenarios.

Recommend setting `exitedTimestamp` flag only when the conditionals are true within `exit()`

Splidge (Reality Cards) confirmed and resolved:

> fixed here

[L-18] `Test` function left behind can expose order book

Submitted by OxRajeev

The `getBid()` order book function is noted in its Natspec @dev comments as "@dev just to pass old tests, not needed otherwise @dev to be deleted once tests updated" but is left behind here.

This function could externally expose orderbook ordering (prev/next linked list) for malicious contracts to potentially time or price bids to its advantage.

Recommend removing the function as noted.

Splidge (Reality Cards) disputed:

> Remove function as noted.

> It is to be removed one the tests are updated. However they are not yet, so it isn't to be removed yet.

> It is simple enough for other tools to determine the order of the orderbook without this, the frontend manages this and displays the information to the users. This is not a vulnerability, but useful information for users to have at their disposal.

dmvt (Judge) commented:

> The issue as I see it is that this exposes orderbook order to other contracts, not just other tools. This opens up some potentially unwanted vectors if left in. IMO, you should have removed this prior to the contest or specifically commented that it should be ignored during the contest. Another potential approach would be to have a testing contract which adds this functionality for testing but does not get automatically deployed to production. If you forget about it and leave it in, it does represent a small risk.

[L-19] Shadowing Local Variables found in `RCOrderbook.sol`

Submitted by maplesyrup (heiho1 and thisguy__)

According to the Slither-analyzer documentation, shadowing local variables is naming conventions found in two or more variables that are similar. Although they do not pose any immediate risk to the contract, incorrect usage of the variables is possible and can cause serious issues if the developer does not pay close attention.

It is recommended that the naming of the following variables should be changed slightly to avoid any confusion.

Recommended mitigation steps:

1. Clone Project Repository
2. Run Project against Hardhat network; compile and run default test on contracts.
3. Installed slither analyzer: https://github.com/crytic/slither
4. Ran [$ slither .] against `RCOrderbook.sol` and all contracts to verify results

Splidge (Reality Cards) confirmed and resolved:

> fixed here

[L-20] `totalNftMintCount` can be replaced with ERC721 `totalSupply()`

Submitted by paulius.eth

I can't find a reason why `totalNftMintCount` in `Factory` can't be replaced with ERC721 `totalSupply()` to make it less error-prone. As `nfthub.mint` issues a new token it should automatically increment `totalSupply` and this assignment won't be needed:

```
totalNftMintCount = totalNftMintCount + _tokenURIs.length;
```

> Also in function `setNftHubAddress` you need to manually set `_newNftMintCount` if you want to change `nfthub` so an invalid value may crash the system. `totalSupply()` will eliminate `totalNftMintCount` and make the system more robust.

Recommend replacing `totalNftMintCount` with nfthub `totalSupply()` in `Factory` contract.

mcplums (Reality Cards) commented:

> This is a good one, would indeed make it less error prone

Splidge (Reality Cards) confirmed and resolved:

> Sometimes what looks like a small fix just takes you all the way down the rabbithole, this was one of them. totalSupply() isn't included by default so I had to import the Enumerable extension. fixed here

[L-21] `RCTreasury.addToWhitelist()` will erroneously remove user from whitelist if user is already whitelisted

Submitted by jvaqa, also found by 0xRajeev and [s1m0] (https://twitter.com/smonica)

`RCTreasury.addToWhitelist()` will erroneously remove user from whitelist if user is already whitelisted

The comments state that calling `addToWhitelist()` should add a user to the whitelist.

However, since the implementation simply flips the user's whitelist bool, if the user is already on the whitelist, then calling `addToWhitelist()` will actually remove them from the whitelist.

Since batch `AddToWhitelist()` will repeatedly call `addToWhitelist()` with an entire array of users, it is very possible that someone could inadvertently call `addToWhitelist` twice for a particular user, thereby leaving them off of the whitelist.

If a governor calls `addToWhitelist()` with the same user twice, the user will not be added to the whitelist, even though the comments state that they should.

Recommend changing `addToWhitelist` to only ever flip a user's bool to true. To clarify the governor's intention, create a corresponding `removeFromWhitelist` and `batchRemoveFromWhitelist` which flip a user's bool to false, so that the governor does not accidentally remove a user when intending to add them.

Also recommend changing `isAllowed[_user] = !isAllowed[_user];` TO `isAllowed[_user] = true;` , and adding this:

```
/// @notice Remove a user to the whitelist
function removeFromWhitelist(address _user) public override
    IRCFactory factory = IRCFactory(factoryAddress);
    require(factory.isGovernor(msgSender()), "Not authorisec
    isAllowed[_user] = false;
}

/// @notice Remove multiple users from the whitelist
function batchRemoveFromWhitelist(address[] calldata _users)
    for (uint256 index = 0; index < _users.length; index++)
        removeFromWhitelist(_users[index]);
    }
}
```

Splidge (Reality Cards) acknowledged but disagreed with severity:

> The whitelist is only really for the initial beta phase so we aren't going to be putting more time and effort into changes here. I think the severity can be

> reduced as the whitelist only limits access to `deposit()`, if a user was added and then erroneously removed they can still partake in the events and withdraw their winnings. It is only limiting their entry into the system.

Splidge (Reality Cards) confirmed:

> An update, We now implement OpenZeppelin AccessControl.sol, so whitelisting is now either granting or revoking the role WHITELIST.

dmvt (Judge) lowered severity from 2 to 1:

> I don't think this is a medium severity issue as calling it a third time will toggle it back on. That said, the naming is confusing and could result in upset users / some reputational damage, particularly if the sponsor changes their mind and keeps this in place past beta. Changing severity to 1

[L-22] Unbounded iteration on `_cardAffiliateAddresses`

Submitted by **cmichel**

The `Factory.createMarket` iterates over all `_cardAffiliateAddresses`.

The transactions can fail if the arrays get too big and the transaction would consume more gas than the block limit. This will then result in a denial of service for the desired functionality and break core functionality.

Recommend performing a `_cardAffiliateAddresses.length == 0 || _cardAffiliateAddresses.length == tokenUris.length` check in `createMarket` instead of silently skipping card affiliate cuts in `Market.initialize`. This would restrict the `_cardAffiliateAddresses` length to the `nftMintingLimit` as well.

Splidge (Reality Cards) confirmed and resolved:

> implemented here

[L-23] `uberOwner` cannot do all the things an owner can

Submitted by **cmichel**

The `uberOwner` cannot do the same things the owner can. They can "only" set the reference contract for the market.

The same ideas apply to `Treasury` and `Factory`'s `uberOwner`.

The name is misleading as it sounds like the uber-owner is more powerful than the owner.

Recommend that `Uberowner` should at least be able to set the owner if not be allowed to call all functions that an `owner` can. Alternatively, rename the `uberOwner`.

mcplums (Reality Cards) confirmed:

> I like this! Is not too important, but can't hurt to have uber owner able to change the owner.

Splidge (Reality Cards) commented:

> I will come back to this issue if time allows. Ownable.sol has been made such that you can't override `transferOwnership()` or the `onlyOwner` modifier. This means the next best option would be changing to AccessControl.sol which is more effort than I think the benefit warrants given our current timescale.

[L-24] Dangerous toggle functions

Submitted by **cmichel**

Usually one tries to avoid toggle functions in blockchains, because it could be that you think that the first transaction you sent was not correctly submitted (but it's just pending for a long time), or you might even be unaware that it was already sent if multiple roles can set it (like with `changeMarketApproval` / `onlyGovernors`) or if it's an msig.

This results in potentially double-toggling the state, i.e, it is set to the initial value again. Some example functions: `changeMarketCreationGovernorsOnly`, `changeMarketApproval`, and the ones that follow. The outcome of toggle functions

is hard to predict on blockchains due to the very async nature and lack of information about pending transactions.

Recommend using functions that accept a specific value as a parameter instead.

Splidge (Reality Cards) confirmed

[L-25] The `domainSeperator` is not recalculated after a hard fork happens

Submitted by shw

The variable `domainSeperator` in `EIP712Base` is cached in the contract storage and will not change after the contract is initialized. However, if a hard fork happens after the contract deployment, the `domainSeperator` would become invalid on one of the forked chains due to the `block.chainid` has changed.

Recommend consider using the implementation from OpenZeppelin, which recalculates the domain separator if the current `block.chainid` is not the cached chain ID.

Splidge (Reality Cards) confirmed:

> The OpenZeppelin implementation can't be used because of the contracts uses a proxy pattern and so can't use the constructor in the OpenZeppelin version. Instead I have taken the same method OpenZeppelin use to get a new `domainSeperator`, here

Splidge (Reality Cards) commented:

> I've noticed that there is an upgradable version of the OpenZeppelin metaTx contracts. I'll try and work on using them instead of the fix used above.

Non-Critical Findings

- [N-01] Use immutable keyword
- [N-02] improve readability of 1000000

- [N-03] 1000 as a constant
- [N-04] Camel case function name
- [N-05] Add comment to not obvious code in withdrawDeposit
- [N-06] event WithdrawnBatch is not used
- [N-07] Missing events in multiple functions
- [N-08] functions safeTransferFrom and transferFrom are too similar
- [N-09] contract RCTreasury does not use nfthub and setNftHubAddress
- [N-10] circuitBreaker overrides the state
- [N-11] Redundant `require()` statement in RCFactory.createMarket()
- [N-12] prevent bids in WINNER*TAKES*ALL when it is no longer possible to win
- [N-13] questionFinalised is redundant
- [N-14] Calculation imprecision when calculating the reaming cut
- [N-15] Unused return value from orderbook.findNewOwner() and treasury.payRent()
- [N-16] Use Mode instead of uint in RCFactory to make code much more readable
- [N-17] timestamp
- [N-18] Unused named return values are misleading and could lead to errors
- [N-19] maxContractBalance can be bypassed

Gas Optimizations

- [G-01] Gas inefficiency with NativeMetaTransaction and calldata
- [G-02] _realitioAddress not used
- [G-03] Checks for enum bounds
- [G-04] costly-loop
- [G-05] Redudant calculations in `payRent` when marketBalance < `_amount`
- [G-06] Gas optimizations - Duplicated state variable
- [G-07] Gas Optimizations - Use storage or memory to reduce reads
- [G-08] Gas optimizations - Remove isMarket from RCMarket

- [G-09] Redundant allowance and balance checks
- [G-10] external-function

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top