QuillAudits

Audit Report
March, 2022

For

Melodity

Contents

Overview 01

Scope of Audit 01

Checked Vulnerabilities 02

Techniques and Methods 03

Issue Categories 04

Issues Found 05

High Severity Issues 05

Medium Severity Issues 09

Low Severity Issues 11

Informative Issues 14

Functional Testing 19

Closing Summary 20

Overview

Melodity

Melodity is bringing the old fashioned music industry to the 21st century connecting it to the awesome world of blockchain technology and decentralization. Our journey toward the Melodity Ecosystem is ongoing and we are glad to see so many of you have already joined us on the road!.

Scope of the Audit

The scope of this audit was to analyze the Melodity smart contract's codebase for quality, security, and correctness.

The Source code for the Audit is Taken from:-
https://github.com/Do-inc/melodity-stacking

Commit: b37c965b7b0e4df69317a1eabc6fae7b31af7cea

Fixed In: 20d96a42dee5196093adbad3bf769923b137437d

Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC20 transfer() does not return boolean
- ERC20 approve() race
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis
In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis
Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis
Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption
In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit
Mythril, Slither, SmartCheck, Surya, Solhint.

Issue Categories

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

Risk-level	Description
High	A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.
Medium	The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.
Low	Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.
Informational	These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

Type	High	Medium	Low	Informational
Open	0	0	0	0
Acknowledged	0	0	0	0
Closed	6	4	8	8

Issues Found – Code Review / Manual Testing

High severity issues

1. **Governance token can be transferred without any whitelisted & blacklisted check.**

   MelodityGovernance.sol is missing an overridden transfer function which should have isWhitelisted and isBlacklisted modifier to check whether the recipient is allowed to receive the tokens or not. Because of this gMELD can be traded freely without any restriction.

   Recommendation
   To have a overridden transfer function like transferFrom [#L83] instead of using the ERC20 function directly. Example -

```
function transfer(
      address recipient,
      uint256 amount
  )
      public
      override
      isWhitelisted(recipient)
      isBlacklisted(recipient)
      returns (bool)
{
    return ERC20.transfer(sender, recipient, amount);
}
```

   Status: Fixed

2. **Funds would get locked for perpetuity during blind auctions.**

   contracts/Marketplace/BlindAuction.sol [#L107] reveal() function allows the user to reveal its bid and transfer the funds back to the user if the bid is fake. Current contract design assumes that user would always call the reveal method during the reveal period to place a bid but as per the human nature it can't hold 100% true as there can be several reason that user may fail to call the reveal() method during the reveal duration that causes the funds of the user get locked for perpetuity because withdraw() function wouldn't work as user would not be registered in the pendingReturns mapping as placeBid() never get called for that affected user.

Recommendation:
Better to have a skim or reap function that will allow reclaiming the sent funds by the msg.sender after the auction gets concluded, This would allow them to reclaim funds even if the user forgot to reveal its bids during the reveal period.

Status: Fixed

3. Reward pool gets exhausted quicklier and locks deposited funds for perpetuity

contracts/Stacking/MelodityStacking.sol [#L381]

```
poolInfo.rewardPool -= meldToWithdraw;
```

meldToWithdraw is the amount that consists of principal & reward both while it gets subtracted from the poolInfo.rewardPool to reduce the reward pool instead of only subtracting the reward part of the withdrawal amount.

This would lead to the exhaustion of the rewardPool variable quicklier then expected and deposited MELD funds would remain in contract for perpetuity as there is no function that would allow to transfer those funds out of the contract.

Recommendation:
poolInfo.rewardPool only get updated with reward value i.e

```
poolInfo.rewardPool -= meldToWithdraw - _amount
```

This would transfer the principal amount from the contract balance and reward would get deducted from the reward balance.

Status: Fixed

4. **Inefficient algorithm to refresh the receipt value that may make it uncallable because of high gas usage.**

   contracts/Stacking/MelodityStacking.sol [#L477]

   eraInfos is an unbounded array that would drain the block gas limit if the eraInfos length would get inappropriate. The other downside of the current implementation is that it always iterates the eraInfos from 0 even if the era got passed already as there is no information stored in the contract that can tell which era index already got passed that makes this more function more gas consumable and more prone to gas exhaustion which cause a ripple effect and make deposit() and withdraw() function uncallable as well.

   Recommendation
   Better to have an upper limit to the eraInfos array length so it would never get unbounded, It can be decided by doing some gas analysis.

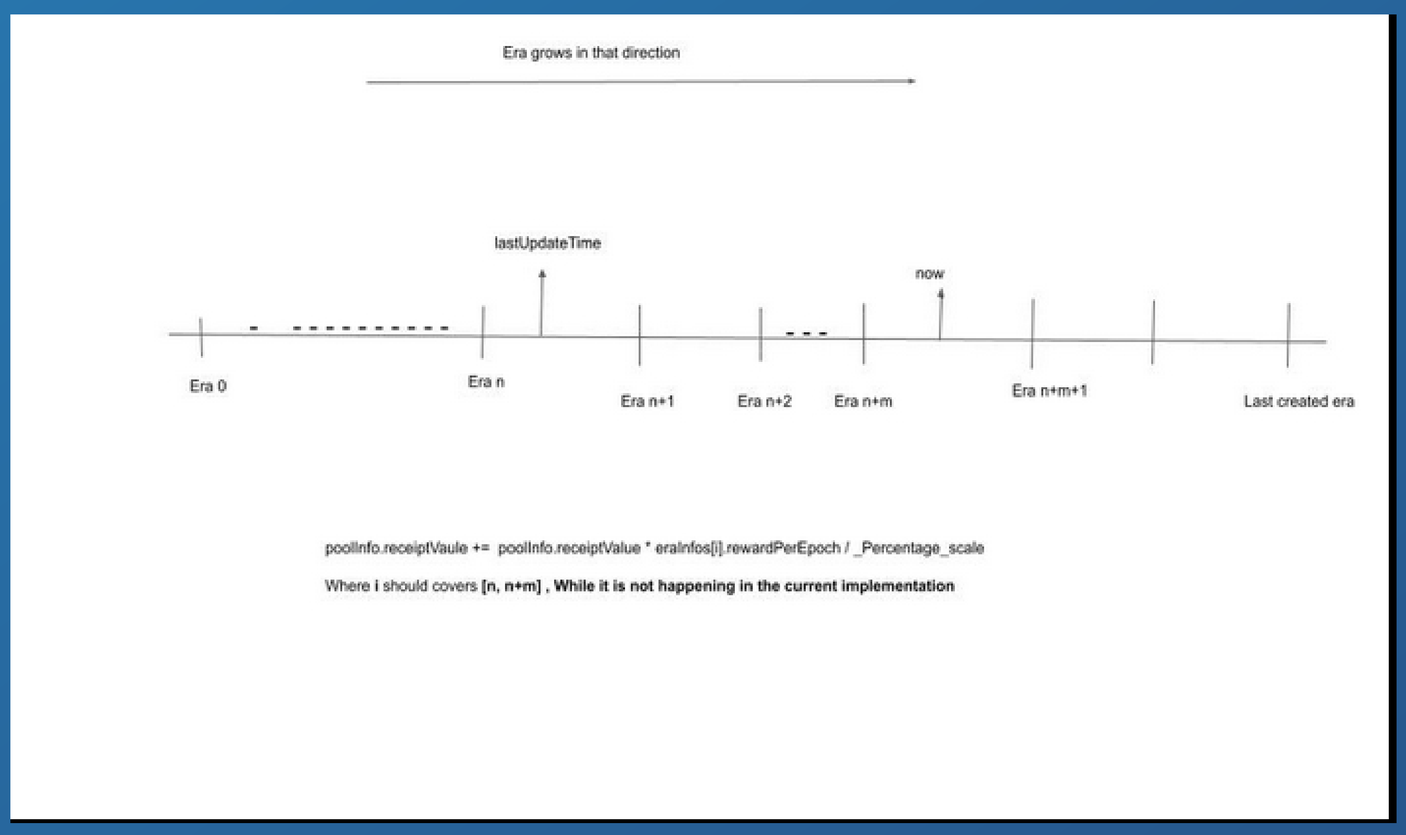
   And have a storage variable that will store the last era index that gets covered by the refreshReceiptValue function call.

   Status: Fixed

5. **Incorrect receiptValue calculation**

   contracts/Stacking/MelodityStacking.sol [#L543]

   If now > lastUpdateTime and now is also not at the same era then it means now lies into the next or another era, which would have startTime > lastUpdateTime that made it fall under the else [#L527] statement, So there is a possibility that now value lies after couple of eras so to calculate right receiptValue for the current timestamp code has to parse those eras and calculate the respective receiptValue using the different eraInfos[eraIndex].rewardFactorPerEpoch whilst at [#L543] poolInfo.receiptValue get calculated on the basis of current era index instead of parsing the eras whose endTime is less than the now.



There is also an edge case that does not get covered that now would exceed the last eraInfo endTime and as per the current implementation we would still have validEraFound = true and it will not create new eras and update the receipt value accordingly.

Current implementation is also not updating the poolInfo.lastReceiptUpdateTime when else code block gets executed.

Recommendation
Should add a check that whether the now > eraInfo[eraInfo.length].endtime then set validEraFound = false again to avoid the edge case depicted above.

Status: Fixed

6. **Recursive nature of the refreshReceiptValue() function would make it gas exhaustive and make it uncallable.**

   Theoretically it is possible that now is in the very distant future and refreshReceiptValue() would not be able to find the valid era that would trigger the _triggerErasInfoRefresh() at [#L557] and theoretically it may take several recursive operations to find the correct era because that function may cause gas exhaustion and inability to call.

   Recommendation
   To set an upper limit on the era counts by doing some gas analysis instead of making free flow.

   Status: Fixed

Medium severity issues

7. Centralization risk in governance model.

contracts/DAO/MeoldityDAO.sol

[#L20] Governance vote quorum fraction is set to 4 which means 4 % quorum gets set that leads to centralization risk as 4 % supply can easily be owned by the owner of the project or team members so it is highly centralized as a governance.

Recommendation
Our recommendation is to set at least 33% vote quorum so it can be sufficiently decentralized to use.

Status: Fixed

8. Inappropriate way of deduction of the withdrawFee.

contracts/Stacking/MelodityStacking.sol [#L390]

As per the current implementation fee would only be deducted when staker tries to withdraw the staking amount before the lastAction + feeInfo.withdrawFeePeriod while lastAction gets always re-written when user deposits the funds init.

This approach would make users hesitate to re-enter into the staking pool as they have to wait again for withdrawFeePeriod to pass out so staker can get the full benefit.

Recommendation
Better approach would be to use the amount of weighted time as the stackersLastDeposit time instead of the current time. So stackersLastDeposit shifted according to how much new amount got staked instead of assigning the block.timestamp.

```
function _updateStakeLastDate(address account, uint256 amt) internal {
        uint256 prevDate = stackersLastDeposit[account];
        uint256 balance = stackingReceipt.balanceOf(account);

        // stakeLastDate + (now - stakeLastDate) * (amt / (balance + amt))
        // NOTE: prevDate = 0 implies balance = 0, and equation reduces to now.
        uint256 newDate = (balance + amt) > 0
            ? prevDate.add(block.timestamp.sub(prevDate).mul(amt).div(balance + amt))
            : prevDate;

        stackersLastDeposit[account] = newDate;
    }
```

Status: Fixed

9. **Exhaustion status is not updating even if the reward pool amount gets increased.**

   contracts/Stacking/MelodityStacking.sol [#L639]

   Reward pool value gets increased using the increaseRewardPool() function while it is not updating the poolInfo.exhausting variable even if the value of the reward pool value is higher than 1 Million. So it will give false indication that reward pool value is less than 1 Million and allow owner to call dimissionWithdraw().

   Recommendation
   Should update the poolInfo.exhausting = false if the balance of the rewardPool is greater than 1 Million.

   Status: Fixed

10. **Minting of stackPanda allow to send NFT to non- ERC 721 Receiver.**

    contracts/StackingPanda.sol [#L75]

    mint() function is using the _mint() function internally instead of using the safeMint() that checks whether the receiver is IERC721Receiver compliant or not that allows to transfer the NFT to the address that is not compliant with ERC721Receiver, but as per the current implementation NFT only get minted to the owner of the contract so it is not considered as the high severity issue.

Recommendation
Use safeMint() at [#L75] instead of _mint().

Status: Fixed

Low severity issues

11. **Incorrect week calculation in number of blocks for different EVM based chains.**

contracts/DAO/MeoldityDAO.sol [#L16]

Voting period is set to 1 week in the block number terms but block duration varies for different EVM based blockchain so if the same contract get deployed on the other chain like Fantom or Ethereum then voting period length would be less than or greater than 1 week as per the chain selected.

Recommendation
To reduce the chance of error it is better to provide voting period length as the constructor param of MelodityDAO contract instead of hardcoding it.

Status: Fixed

12. **Missing event misleads the offchain accounting.**

contracts/Marketplace/Auction.sol[#L151] & contracts/Marketplace/BlindAuction.sol[#L201]

All funds are sent to the beneficiary when the beneficiary and royaltyReceiver are same but there is no emission of RoyaltyPaid event as it is happening at [#L160] or [#L201]. Even if both the addresses are the same it doesn't mean the royalty did not get paid by the marketplace strategy.

Recommendation
Should calculate the royalty earning very similar that is happening at [#L155] or [#L201] and use that value to emit the RoyaltyPaid event in if block i.e [#L149 - #L152].

Status: Fixed

13. **Inefficient use of rotate() function in the entire codebase.**

prng.rotate() gets used in almost every function of the contracts, The rationale seems to update the seed value to make it unpredictable as per the operations performed but it is not an efficient way to do it as calculating hash is an expensive operation that gets performed during the rotate() call and the output never get used in the parent function.

Recommendation
Better to have a separate function in the PRNG contract that will only update the seed value instead of calculating the keccak256 hash as the return value.

Status: Fixed

14. **Incorrect interface id of ERC20 metadata.**

contracts/Stacking/MelodityStacking.sol [#L19] has incorrect interface id for the ERC20 metadata. It should be 0xa219a025 as per the type(IERC20Metadata).interfaceId but it is assigned to 0x942e8b22.

Recommendation
Either it can be completely removed as it never gets used in the contract or it should be assigned to the correct value.

Status: Fixed

15. **Inefficient storage usage in reading maximum and minimum Fee values.**

contracts/Stacking/MelodityStacking.sol [#L167][#L168] FeeInfo struct stores maxFeePercentage & minFeePercentage but these two values never get changed during the entire lifecycle of the MelodityStacking contract. Because of the nature of storage,reading those values costs gas that can be saved by marking those variables constant or immutable.

Recommendation
Remove maxFeePercentage & minFeePercentage from the FeeInfo struct and create the separate immutable variables in the contract scope to avoid charging the gas cost while reading the values of the variables

Status: Fixed

16. **Unnecessary sanity checks.**

    There are multiple places in the contracts/Stacking/MelodityStacking.sol where unnecessary require statements get used while those checks have already been covered in the external contract function calls.

    contracts/Stacking/MelodityStacking.sol [#L287 - #L288] - same checks covered in transferFrom function call happening at [#L294].

    contracts/Stacking/MelodityStacking.sol [#L319 - #L320] - same checks covered in safeTransferFrom function call happening at [#L323].

    contracts/Stacking/MelodityStacking.sol [#L364 - #L371] - same checks covered in burnFrom function call happening at [#L376].

    contracts/Stacking/MelodityStacking.sol [#L633 - #L634] - same checks covered in transferFrom function call happening at [#L636].

    Recommendation
    Better to remove those sanity checks as those are covered in other external function calls within the same parent function and save some gas.

    Status: Fixed

17. **Improvements in the refreshReceiptValue function implementation to make it more gas efficient.**

    contracts/Stacking/MelodityStacking.sol[#L489] In for loop eraInfos.length variable gets read again and again from the storage so it is better to store it into the memory variable and use it into the for loop for less gas consumption.

    contracts/Stacking/MelodityStacking.sol[#L506 - #L510] can be easily re-written as

```
diff = (_now - lastUpdateTime) / _EPOCH_DURATION
```

Because the solidity division works on the floor mechanism so it will always remove the remainder automatically.
And lastReceiptUpdateTime can be represented as

```
poolInfo.lastreceiptUpdateTime = lastUpdateTime + diff *
_EPOCH_DURATION
```

contracts/Stacking/MelodityStacking.sol[#L523] Instead of assigning the eraInfos.length to break the for loop, Better to use the break statement to stop the current loop.

Status: Fixed

18. **Accessing the array index that may be out of bound.**

    contracts/Masterchef.sol [#L105 - #L106]

    Here trying to access the value at index lastPandaId + 1 and there is a high possibility that the value doesn't exist at the given index and that would lead the array index out of bound.

    Recommendation
    Better to check array length first before accessing the given index value.

    Status: Fixed

Informational issues

19. **Compiler error while compiling MelodityDAO.**

    contracts/DAO/MelodityDAO.sol [#L12] is causing the compiler error because ERC20Votes contract doesn't exists in the imported contracts, The cause of this compiler error is that OZ's version is not pinned to 4.4.1, because of that npm installs the higher OZ version and break the build of the contracts.

    Recommendation
    In package.json [#L19] replace it with "@openzeppelin/contracts": "4.4.1",
    It will pinned the OZ dependency.

    Status: Fixed

20. Unnecessary safeTransferFrom function.

contracts/DAO/MelodityGovernance.sol [#L97] Have safeTransferFrom() function which is no difference from transferFrom() function at [#L83]. It can create confusion to the developers and lead them to use the safeTransferFrom() by assuming it is a function of SafeERC20.safeTransferFrom().

Recommendation
It is better to completely remove the safeTransferFrom() function from the MelodityGovernance.sol.

Status: Fixed

21. Use `interfaceId` instead of using hard coded bytes4 value to avoid human error.

contracts/Marketplace/Marketplace.sol [#L57] & [#L66] having the interface Id that is calculated as described in the code comments and stored as the hex in the constant variable. As described, the current setup is prone to human errors and it is hard to formally verify whether the calculated interface ids are correct or not.

Recommendation
Use type(IERC721).interfaceId and type(IERC721Metadata).interfaceId instead of 0x80ac58cd & 0x5b5e139f respectively.

Status: Fixed

22. Initialized royalty can be cheaper in terms of gas consumption.

contracts/Marketplace/Marketplace.sol [#L249] returning royalty struct by reading directly from the storage that is performing 4 extra sloads.

Recommendation
It can easily avoided by storing the Royalty struct in memory at [#L234] and assign the same memory struct to the royalties mapping and use the same to return the royalty struct.

```
        Royalty memory cachedRoyalty = Royalty({
            decimals: 18,
            royaltyPercent: _royaltyPercent, // the provided value *MUST* be padded to 18 decimal
positions
            royaltyReceiver: _royaltyReceiver,
            royaltyInitializer: _royaltyInitializer
          });

        royalties[royaltyIdentifier] = cachedRoyalty

        emit RoyaltyUpdated(
            _nftId,
            _nftContract,
            _royaltyPercent,
            _royaltyReceiver,
            _royaltyInitializer
        );

        return cachedRoyalty;
```

Status: Fixed

23. **Better naming convention**

    contracts/Stacking/MelodityStacking.sol [#L198] existingErasInfos is the variable name used to hold the number of eraInfos stored in the contract or in other words eraInfos array length, But it is not cleared as per the variable name that it holds.

    Recommendation
    It can be renamed as existingEraInfoCount or noOfExistingEras.

    Status: Fixed

24. **Avoid duplicacy of the code to minimize the human error.**

    contracts/Stacking/MelodityStacking.sol [#L211- #L229] & [#L240 - #L254] has the same code that gets deduplicated by introducing an internal function to avoid human error and make code more readable.

    Recommendation
    [#L211 - #L235] Can rewrite as below

```
else if(existingErasInfos > k && eraInfos[k].startingTime > block.timestamp) {
    eraInfos[k] = getNewEraInfo(k)
    // as an era was just updated increase the i counter
    i++;
    // in order to move to the next era or start creating a new one we also need to increase
    // k counter
    k++;
}
```

Where getNewEraInfo() function definition look like this

```
function getNewEraInfo(uint256 k) internal return(EraInfo memory) {
    uint256 lastTimestamp = k == 0 ? poolInfo.genesisTime : eraInfos[k - 1].startingTime +
eraInfos[k - 1].eraDuration;
    uint256 lastEraDuration = k == 0 ? poolInfo.genesisEraDuration : eraInfos[k - 1].eraDuration;
    uint256 lastEraScalingFactor = k == 0 ? poolInfo.genesisEraScaleFactor : eraInfos[k -
1].eraScaleFactor;
    uint256 lastRewardScalingFactor = k == 0 ? poolInfo.genesisRewardScaleFactor : eraInfos[k -
1].rewardScaleFactor;
    uint256 lastEpochRewardFactor = k == 0 ? poolInfo.genesisRewardFactorPerEpoch :
eraInfos[k - 1].rewardFactorPerEpoch;

    uint256 newEraDuration = k != 0 ? lastEraDuration * lastEraScalingFactor /
_PERCENTAGE_SCALE : poolInfo.genesisEraDuration;

    return EraInfo({
        // new eras starts always the second after the ending of the previous
        // if era-1 ends at sec 1234 era-2 will start at sec 1235
        startingTime: lastTimestamp + 1,
        eraDuration: newEraDuration,
        rewardScaleFactor: lastRewardScalingFactor,
        eraScaleFactor: lastEraScalingFactor,
        rewardFactorPerEpoch: k != 0 ? lastEpochRewardFactor * lastRewardScalingFactor /
_PERCENTAGE_SCALE : poolInfo.genesisRewardFactorPerEpoch
    })
```

Similar code can be used at [#L240 - #L256].

Status: Fixed

25. Improvement in events definition for better & efficient offchain usage.

In contracts/Stacking/MelodityStacking.sol Many event params can be indexed to support better offchain filtering.

[#L303] - No need to emit the timestamp as the event itself has the same timestamp as the block so it is just a waste of the gas.

Also recommend to have an indexed account address in the Deposit event to make it easy to filter off chain.
[#L341] - Account can be indexed for better off chain filtering. And should emit the index of the stackedNFT array as well so it can be used during the withdrawWithNFT() function call.

Status: Fixed

26. Incorrect comment

contracts/Masterchef.sol [#L54 - #L61] & [#L63-#L67]

It says “Deploy the Pseudo Random Number Generator using the create2 method” whilst it is deployed using the new keyword that doesn’t use create2 opcode instead use create opcode.

Recommendation
Replace create2 with new.

Status: Fixed

Functional Testing Results

The complete functional testing report has been attached below:
Melodity test case

Some of the tests performed are mentioned below:

- ☑ should be able to add panda identification info.
- ☑ should be able to mint stacking pandas.
- ☑ should be able to list for the sale using a simple auction.
- ☑ should be able to list for the sale using a blind auction.
- ☑ should be able to blind bid on a listed sale.
- ☑ should be able to reveal blind bids on a listed sale.
- ☑ should be able to end blind auctions and move NFT to auction winners.
- ☑ should be able to withdraw bid funds that don't get fulfilled.
- ☑ should be able to reap funds out of the blind auction.
- ☑ should be able to deposit funds in a stacking contract.
- ☑ should be able to deposit NFT and funds in the stacking contract.
- ☑ should be able to withdraw the funds and earn rewards.
- ☑ should be able to charge a fee during withdrawal if it happens before a threshold time.
- ☑ should be able to withdraw NFT.
- ☑ should be able to refresh the receipt value.
- ☑ should be able to increase the reward pool.
- ☑ should be able to refresh era info.
- ☑ should be able to update the era configuration values.
- ☑ should be able to dismission withdraw.

Status: Fixed

Closing Summary

Some issues of High severity, Medium severity were found. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

All the issues has been Resolved by the Auditee.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the Melodity. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the Melodity team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

Audit Report
March, 2022

For

Melodity

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com