Code Review and Security Assessment
For
Dyson Finance

Delivery: November 22, 2023

Prepared For:
Ping Chen | *Dyson Finance*

Prepared by:
Soumen Jana | *HashCloak Inc.*

HashCloak

Table Of Contents

HashCloak

Executive Summary

Dyson Finance is a Dynamic fee AMM market that aims to minimize the risks associated with being a liquidity provider (LP). As such, *Dyson Finance* makes it easier for less sophisticated retail investors to become LPs for AMMs.

From *October 30th, 2023* to *November 22nd*, *2023 the Dyson Finance team* engaged HashCloak for an audit of their Dynamic Fee AMM protocol. The relevant codebase is in the repository: Dyson-Finance-V1, assessed at commit: d5ad1de7ed52e6fa26003e80d02d9ab262f3a90e
The scope of the audit was below files:

- `Dyson-Finance-V1/tree/20231030audit/src/Agency.sol`
- `Dyson-Finance-V1/tree/20231030audit/src/AgentNFT.sol`
- `Dyson-Finance-V1/tree/20231030audit/src/Bribe.sol`
- `Dyson-Finance-V1/tree/20231030audit/src/BribeFactory.sol`
- `Dyson-Finance-V1/tree/20231030audit/src/DYSON.sol`
- `Dyson-Finance-V1/tree/20231030audit/src/Deploy.sol`
- `Dyson-Finance-V1/tree/20231030audit/src/DysonToGo.sol`
- `Dyson-Finance-V1/tree/20231030audit/src/Factory.sol`
- `Dyson-Finance-V1/tree/20231030audit/src/Farm.sol`
- `Dyson-Finance-V1/tree/20231030audit/src/ForeignAgency.sol`
- `Dyson-Finance-V1/tree/20231030audit/src/Gauge.sol`
- `Dyson-Finance-V1/tree/20231030audit/src/GaugeFactory.sol`
- `Dyson-Finance-V1/tree/20231030audit/src/Pair.sol`
- `Dyson-Finance-V1/tree/20231030audit/src/Router.sol`
- `Dyson-Finance-V1/tree/20231030audit/src/sDYSON.sol`

Throughout the audit, we familiarized ourselves with the contracts in scope and sought to understand how the overall *Dyson Finance* protocol works. As we familiarized ourselves with the smart contracts, we focused on finding simple bugs and issues within the codebases. We also focused on finding more complex bugs and issues. We combined our manual analysis skills for the smart contract alongside automated, off-the-shelf tooling such as Slither.

Overall, we found the issues range from medium to informational:

Severity	Number of Findings
Critical	0
High	0
Medium	1
Low	3
Informational	4

HashCloak

Overview

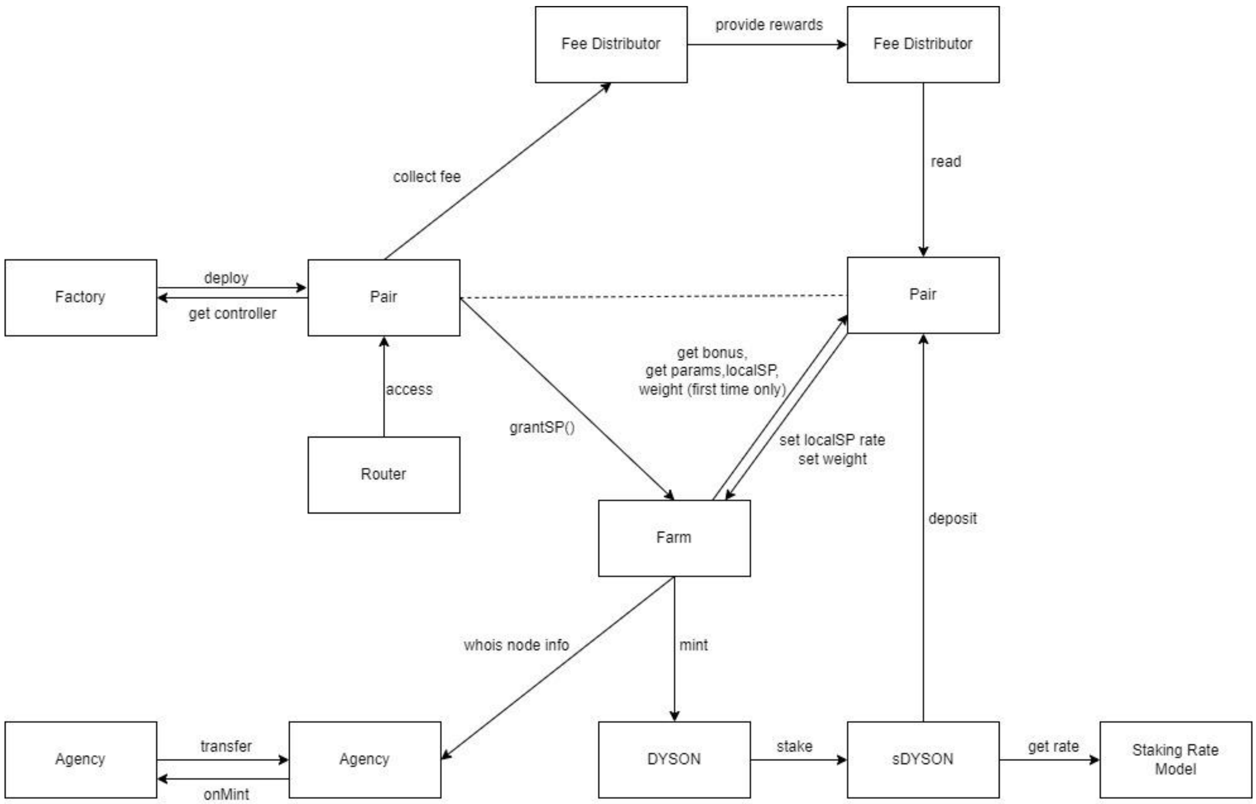
Dyson Finance is a Dynamic fee AMM market in which users can pick a trading pair, then they can choose a maturity date (1, 3, 7, or 30 days), investment amount, and deposit the token in the Pair contract. Post deposit, funds are locked until maturity and users earn instant Points. After maturity, users can withdraw their assets. Redemption of assets is based on fair vs. strike price for optimal returns. If the fair price at redemption is higher then the user will receive the strike price. Conversely, if the fair price at redemption is lower then the user will be receiving the strike price and vice versa. This streamlined process ensures efficient liquidity management and lucrative rewards for *Dyson Finance* users.

The *Dyson Finance* protocol has the following contracts: (see their specification)

- Agency.sol Boost your DYSON holdings – deposit, refer, and earn rewards
- AgentNFT.sol ERC721 extension for Agency, enabling single NFT holdings and transfers
- Bribe.sol Incentivizes sDYSON holders to deposit in specific Gauge
- BribeFactory.sol Bribe is deployed with this factory contract
- DYSON.sol Governance token of *Dyson Finance*
- Deploy.sol Deployer Contract
- DysonToGo.sol Simplifies $DYSN mining participation
- Factory.sol Pair is deployed with this factory contract
- Farm.sol Earn extra rewards, By swapping SP to DYSON
- ForeignAgency.sol Referral system contract for foreign chain
- Gauge.sol Liquidity pool voting contract, deposit sDYSON, earn additional rewards
- GaugeFactory.sol Gauge is deployed with this factory contract
- Pair.sol Fee-based Dyson pair with time-locked notes and liquidity pool mechanisms
- Router.sol Router for Pair contracts enabling swaps, deposits, and withdrawals
- sDYSON.sol Staking $DYSN will get sDYSN and Voting Power

HashCloak

We have made the following diagram to help clarify the variables each contract is in charge of and the important function calls between these contracts.



Methodology

Our audit methodology was as follows:

1. Go over the *Dyson.Finance* documentation provided by *Dyson.Finance* team
2. Based on the documentation, we formulated an initial checklist that combines items for the most common Solidity bugs and our initial ideas on areas in which we believe *Dyson.Finance* might be a cause for concern
3. We had an initial conversation with the *Dyson.Finance* team to go over our checklist
4. Do an initial pass through the *Dyson.Finance* codebase to find the most common Solidity implementation bugs and easier to identify issues in *Dyson.Finance*

5. We then used off-the-shelf automated static analysis tools. In particular, we ran Slither and Mytril
6. We did another pass through the codebase for more complex bugs and areas of concern based on our research and the output of the automated tooling

HashCloak

Findings

DF-1: Potential Failure in DysonToGo Contract Due to Missing Membership NFT

Type: Medium
Files affected: DysonToGo.sol#L234C4-L240C6

Description: The audit has identified a potential issue in the `DysonToGo` contract, where the absence of a membership NFT may fail the line dysonAmount = dysonPool * sp / (spPool + sp); when both `spPool` and `sp` are zero.

Impact: In scenarios where the `DysonToGo` contract lacks a membership NFT, and `spPool` along with `sp` are both zero, the mentioned line will encounter a division by zero error, leading to a revert. This could result in an inability to process withdrawals correctly and potential disruptions in contract functionality.

Suggestion: Implement conditional statements or checks to handle the scenario where spPool and sp are both zero, preventing division by zero errors and maintaining the stability of contract operations.

DF-2: Lack of Event Emission for Sensitive Data Updates

Type: Low
Files affected: DysonToGo.sol

Description: The audit has revealed a notable absence of event emission within the codebase when sensitive data is updated, such as `adminFeeRatio`, `updatePeriod`, etc. Events serve as a critical means of providing transparency and traceability, especially for changes to sensitive information.

Impact: The absence of event emissions for sensitive data updates can hinder the ability to monitor and track changes within the system. This may lead to a lack of accountability, making it challenging to identify the source and reason for modifications to critical data, potentially resulting in security vulnerabilities and operational difficulties.

Suggestion: It is advisable to implement event emission mechanisms for all updates to sensitive data in the codebase. This practice will enhance transparency, accountability, and the ability to monitor and audit changes effectively, ultimately improving the security and reliability of the system.

DF-3: Potential Risk of Damage Due to `onlyOwner` Privileges and Private Key Exposure

Type: Low

Files affected: DYSON.sol , sDYSON.sol

Description: The audit has identified a potential vulnerability in the project where certain functionalities are restricted by the `onlyOwner` modifier. In the event of a private key leak associated with the owner account, unauthorized access to critical project functions may occur, leading to the potential for damage.

Impact: The use of `onlyOwner` privileges, coupled with a possible private key leak, poses a significant risk to the project's security. Unauthorized access to the Owner account for `DYSON.sol` can lead to an unlimited number of token minting by the malicious user, etc.

Suggestion: It is strongly advised to reconsider the use of `onlyOwner` privileges, especially for critical functions. Implement additional security measures such as multi-signature schemes to reduce the risk associated with a potential private key leak.

DF-4: Lack of Clarity in the Usage and Conditions of rescueERC20 Function in DYSON.sol

Type: Low

Files affected: DYSON.sol, sDYSON.sol

Description: The audit has identified a potential ambiguity in the `DYSON.sol` contract, specifically regarding the `rescueERC20` function. The function lacks clear documentation specifying when it can be used and under what conditions.

Impact: The absence of explicit information on the usage and conditions for the `rescueERC20` function may lead to misunderstandings, misuse, or unintended consequences. Users and developers may struggle to determine the appropriate scenarios for invoking this function, potentially jeopardizing the integrity of the contract.

Suggestion: It is recommended to provide comprehensive documentation within the code or accompanying documentation specifying the circumstances under which the `rescueERC20` function should be utilized. Clearly outline the intended use cases, restrictions, and any associated risks to ensure proper understanding and secure implementation.

DF-5: Code coverage should be close to 100%

Type: Informational
Files affected: All Contract In Scope

Description: Code coverage is a measure used to describe how much of the source code is executed during the automated test suite. A system with high code coverage, measured as lines of code executed, has a lower chance of containing undiscovered bugs. Some of the contracts have only ~50% coverage, e.g.: `DYSON.sol`, `DysonToGo.sol`, `TreasuryVester.sol`, `FeeDistributor.sol`, `AddressBook.sol`, etc.

Impact: Adequate test coverage and regular reporting are essential processes in ensuring the codebase works as intended. Insufficient code coverage may lead to unexpected issues and regressions arising due to changes in the underlying smart contract implementation.

Suggestion: Ensure the coverage report produced via `forge coverage` covers all functions within *Dyson Finance*'s smart contract.

DF-6: Absence of Consistent Comments in the Codebase

Type: Informational

Files affected: All Contract In Scope

Description: The audit has identified a lack of consistent comments within the codebase, indicating insufficient documentation and explanatory notes for various sections of the code. E.g.: `gaugeWithdraw()`, `bribeClaimRewards()`, `update()`, etc. function doesn't have any comments about its use cases.

Impact: The absence of consistent comments can impede code comprehension, making it challenging for developers to understand the purpose, functionality, and potential risks associated with specific code segments. This lack of clarity may result in increased development time, a higher likelihood of introducing errors, and difficulties in maintaining and updating the code.

Suggestion: To enhance code readability and maintainability, it is recommended to implement consistent commenting practices throughout the codebase. Comments should include explanations of complex logic, the rationale behind design choices, and any potential risks or considerations. This will facilitate easier collaboration among developers, streamline future updates, and contribute to a more robust and comprehensible codebase.

DF-7: Redundant Zero Address Checks in Input Validation

Type: Informational
Files affected: All Contract In Scope

Description: The audit has identified instances in the code where an address is provided as input and subsequently checked for being zero. This redundant zero address check may lead to unnecessary gas consumption.

Impact: The redundant zero address checks in `constructor`, `transferOwnership`, `gaugeDeposit`, etc in `DysonToGo.sol` and similar for the rest of the contracts can contribute to increased gas consumption, potentially affecting transaction costs and overall efficiency. While these checks serve the purpose of ensuring input validity, they may be redundant if an equivalent validation is performed at the front end.

Suggestion: It is recommended to evaluate the necessity of zero address checks in cases where equivalent validations are conducted at the front end. Removing redundant zero address checks in such scenarios can contribute to gas savings and enhance the efficiency of the contract execution.

DF-8: Non-Immutable State Variables Unchanged Post-Deployment

Type: Informational
Files affected: All Contract In Scope

Description: The audit has observed state variables in the code like `gauge` variable in the `Bribe` contract; `agency`, `dyson`, `factory`, `farm`, `rateModel`, `router`, `sdyson` variables in the `Deploy` contract; `owner` variable in the `FeeDistributor` contract and `icoToken`, `token0`, `token1`, `totalSupply` variables in the `ICO` contract; `token`, `vestingAmount`, `vestingBegin`, `vestingCliff`, `vestingEnd` variables in the `TreasuryVester` contract that remain static and unaltered following deployment. Despite their lack of modification, these variables are not declared as immutable.

Impact: Failing to declare non-changing state variables as immutable can result in unnecessary gas consumption. Declaring variables as immutable informs the compiler that their values will not change, enabling optimization and reducing gas costs.

Suggestion: To optimize gas usage, consider declaring state variables that remain constant after deployment as immutable. This informs the compiler of their static nature, resulting in more efficient contract deployment and reduced gas consumption.