Dyson Finance

Smart Contract Security Assessment

October 13, 2022

ABSTRACT

Dedaub was commissioned to perform a security audit of parts of the Dyson Finance protocol.

The Dyson Finance protocol is an investment protocol that aims to provide users with a decentralized alternative to centralized Dual Investment options offered by CEXs. At the core of protocol resides a special AMM (Automated Market-Maker), thus invested funds not only flow transparently, but also increase liquidity in the market. Dyson Finance also incorporates SocialFi and referral elements, like Dyson's Satellite Program.

This audit report covers the contracts of the at the time private repository Gabriel-Dyson/dyson-audit-2022-09-26 of the Dyson Finance protocol, up to commit hash 533839d232403d701ef6ae5d77cb1ba3347ce6fc.

The audit did not consider the financial viability of the protocol in depth. It is apparent that since deposits to a Dyson pool can have different lock periods, there could exist a situation where the pool does not hold enough capital to pay back the deposited amounts plus the agreed premium in whole. The Dyson team was consulted on this matter and indicated that there exists the risk that a pool could go bankrupt. In order to prevent such scenarios, the team indicated that they would carefully tune the basis and halfLife parameters of the pools. Finally, according to the team, the frontend of the app will warn users in case a pool is running out of liquidity and this particular investment risk will also be disclosed to users via the project's documentation.

The full audited contract list is the following:

```
src/Agency.sol
src/AgentNFT.sol
src/Bribe.sol
src/Deploy.sol
```

src/Dyson.sol
src/DysonFactory.sol
src/DysonPair.sol
src/DysonRouter.sol
src/Farm.sol
src/Gauge.sol
src/sDYSON.sol
src/TransferHelper.sol

SETTING & CAVEATS

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than regular use of the protocol. Functional correctness (i.e., issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e., full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification.

Functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing. The scope of the audit includes smart contract code. Interactions with off-chain (front-end or back-end) code are not examined other than to consider entry points for the contracts, i.e., calls into a smart contract that may disrupt the contract's functioning.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: 1) User or system funds can be lost when third-party systems misbehave. 2) DoS, under specific conditions. 3) Part of the functionality becomes unusable due to a programming error.
LOW	Examples: 1) Breaking important system invariants but without apparent consequences. 2) Buggy functionality for trusted users where a workaround exists. 3) Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

There were no critical severity issues.

HIGH SEVERITY:

There were no high severity issues.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	sDYSON functionality can be DOSed	**RESOLVED**

Functions `dysonAmountStaked` and `votingPower` of the sDYSON contract implement a loop over a user's vaults.

```
function dysonAmountStaked(address _user) external view
    returns (uint totalAmount)
{
  uint userVaultCount = vaultCount[_user];
  for(uint i; i < userVaultCount; i++) {
    totalAmount += vaults[_user][i].dysonAmount;
  }
}
```

Even though the operations performed inside the loop are not particularly gas expensive, the possibility of running out of gas always exists when the loop is virtually unbounded. At the same time, the fact that sDYSON::stake allows one to create a vault for any user adds the possibility of DOS attacks, as one could stake tiny amounts of DYSON for the user they want to DOS as many times as needed to make calls to the aforementioned functions really expensive or even infeasible.

M2	Gauge	DISMISSED

A farm's reward rate is applied with a delay of one operation, i.e., if a user deposits sDYSON to a gauge to increase the corresponding farm's reward rate by increasing the total supply of the gauge, this will not happen unless the following `deposit`, `withdrawal` or call to `tick` happens. This happens due to the function `updateRewardRate` being called at the end of `Gauge::tick` while `tick` is called by `updateTotalSupply` before assigning the new total supply (`_totalSupply`) to the `totalSupply` storage variable, which is consulted by `Gauge::nextRewardRate`.

```
function tick() public {
  uint _week = block.timestamp / 1 weeks;

  if (_week > thisWeek) {
    for(uint i = thisWeek; i < _week; ++i) {
      _totalSupplyAt[i] = totalSupply;
    }
    thisWeek = _week;
    updateRewardRate();
  }
}

function updateTotalSupply(uint _totalSupply) internal {
  tick();
  totalSupply = _totalSupply;
}

function nextRewardRate() public view returns (uint newRewardRate) {
  newRewardRate = totalSupply * slope / REWARD_RATE_BASE_UNIT + base;
}

function updateRewardRate() internal {
  try farm.setPoolRewardRate(poolId, nextRewardRate(), weight) {} catch {}
}
```

Thus, reward rate is computed using the old `totalSupply`, as `tick` will try to backfill any missing information about previous weeks with the last known `totalSupply`

before calling `updateRewardRate`. We believe that the call to function `UpdateRewardRate` should be moved to the end of function updateTotalSupply, after the update of `totalSupply`.

LOW SEVERITY:

ID	Description	STATUS
L1	First AP to Dyson swap done by a user does not have a cooldown period.	**DISMISSED**
In the Farm contract, when the `swap()` function is called for the first time by a user, the value of cooldown[user] is zero. This means that the requirement `block.timestamp > cooldown[user]` at the beginning of the function will be satisfied, resulting in the user being able to bypass the cooldown period.		
L2	Gauge resets withdrawal period of pending withdrawals each time `applyWithdrawal()` is called.	**DISMISSED**
In the Gauge contract, the `applyWithdrawal()` function resets the withdrawal period for a user each time the function is called, i.e., every time there is a new withdrawal request. Thus, if the user already has a pending withdrawal, and decides to withdraw some more sDyson tokens from the gauge, the period of the already pending withdrawal is extended as well.		
L3	Truncation of timestamp information	**DISMISSED**
In the DysonPair contract, the `updateFeeRatio0()` and `updateFeeRatio1()` functions store the block.timestamp, which is of type uint256, inside the contract variables lastUpdateTime0 and lastUpdateTime1, which are of type uint64. However, in the `getFeeRatio()` function, the truncated variables are cast to uint256 and subtracted from the block.timestamp. It is possible that in the future, the truncated variables can		

no longer contain the correct block number, resulting in `feeRatio()` giving an incorrect output going forward.		
L4	Infinite approval in sDYSON::migrate could mess up migrations/accounting	**RESOLVED**
Function `sDYSON::migrate` approves the migration of an unlimited amount of sDYSON tokens to the `migration` contract. Even if the `migration` contract is trusted, which is the expected scenario, the unlimited approval could allow messing up the contract state in the unlikely scenario where migration is mistakenly programmed to transfer the whole sDYSON balance of `msg.sender` instead of only the balance of the migrated vault. This would result in all vault subsequent migrations for the same user failing due to insufficient balance, as the whole balance would have been transferred during the first migration, trapping the remaining Dyson tokens of the user.		
L5	Bribe functions could be reentered	**DISMISSED**
Functions `Bribe::addReward` and `Bribe::claimReward` accept and make calls to arbitrary tokens. As there are no reentrancy guards, an adversary could add as reward a malicious token and then use it to reenter any of these functions. We have considered different reentrancy attacks but have not identified any real danger. Still, we would advise the protocol developers to implement reentrancy guards for the aforementioned functions to increase the security of the current and future versions of the code.		
L6	Dyson and sDYSON _mint, _burn and _transfer functions do not check the from and to parameters	**DISMISSED**
The Dyson and sDYSON `_mint`, `_burn` and `_transfer` functions do not implement any checks for the validity of the `from` and `to` parameters provided to them. The following requirements would increase the security of the operations: • _mint: `require(to != address(0), "...")` • _burn: `require(from != address(0), "...")`		

• _transfer: `require(from != address(0) && to != address(0), "...")`		
L7	DysonRouter has no way to revoke token approvals	**RESOLVED**
Function `DysonRouter::rely` can be used by the owner of `DysonRouter` to approve other contracts for the provided tokens. However, there is no implemented functionality that could be used to revoke these approvals.		

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	STATUS
N1	The owner and minters can mint an unlimited number of Dyson tokens.	**DISMISSED**
In the Dyson contract, the `mint()` function can be called by the owner of the contract or by a minter approved by the owner. This allows the owner or the minter to mint an unlimited number of Dyson tokens after deployment.		
N2	Compromise of the owner key of the sDYSON contract can lead to the complete drainage of DYSON funds	**DISMISSED**
The sDYSON::migrate function allows a user to migrate their DYSON and sDYSON to the migration contract, which can only be set by the owner of the sDYSON contract. An attacker could drain the whole DYSON balance of the sDYSON contract as long as they are able to get hold of the sDYSON contract's owner key. That would be possible as sDYSON::migrate does not implement the checks-effects-interactions pattern, i.e., storage variable updates happen after external calls to the migration contract and to the DYSON contract.		

```
function migrate(uint index) external {
  require(migration != address(0), "CANNOT MIGRATE");
```

```
    Vault storage vault = vaults[msg.sender][index];
    require(vault.unlockTime > 0, "INVALID VAULT");
    uint amount = vault.dysonAmount;
    // Dedaub:
    require(IsDYSONUpgradeReceiver(migration).(msg.sender, index) ==
      _MIGRATE_RECEIVED, "MIGRATION FAILED");
    Dyson.safeTransfer(migration, amount);
    _approve(msg.sender, migration, type(uint).max);
    vault.dysonAmount = 0;
    vault.sDYSONAmount = 0;
    vault.unlockTime = 0;
    emit Migrate(msg.sender, index);
  }
```

First, the attacker would set the `migration` contract to a malicious contract that implements `onMigrationReceived(address,uint256)`. The attacker would proceed by depositing an appropriate amount of DYSON to the sDYSON contract and then calling migrate to transfer control to the malicious migration contract that would reenter sDYSON::migrate. Due to the fact that `vault.unlockTime = 0` happens after the external call to `migration` the `require(vault.unlockTime > 0, "INVALID VAULT")` statement cannot prevent the reentrancy and the attacker can drain the whole DYSON balance of the contract by calling `Dyson.safeTransfer(migration, amount)` multiple times.

OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Redundant steps in Deploy contract's constructor	**INFO**
In the Deploy contract's constructor, the sDYSON contract is initially created with the deploy contract as its owner. The constructor then transfers the ownership of the sDYSON contract to the owner specified in the constructor's parameters. This can be simplified by setting the owner of the sDYSON contract directly to the owner specified in the constructor's parameter.		
A2	Zero EXTCODESIZE is not a reliable way of determining whether an address is a contract.	**INFO**
The `AgentNFT::_isContract()` function checks whether an address is a contract by determining whether EXTCODESIZE is zero. Please note that this is not a reliable way of checking whether an address is a contract, since contracts have zero EXTCODESIZE when their constructor is being executed and calls can be made by an attacker at this stage. Currently the `_isContract()` function is used to trigger the onERC721Received hook when an NFT is transferred to a receiver by the sender itself and when a sender has the approval to perform the transfer on behalf of a third party contract, but this should be used cautiously in future updates.		
A3	Several storage variables can be made immutable	**INFO**
There exist several storage variables that are set in the constructor and cannot be modified ever since. These variables can be declared as immutable: • All storage variables of the `Deploy` contract		

A4	Additional requirements in sDYSON::stake and restake	INFO
Function `sDYSON::stake` could implement a require statement that ensures the `to` parameter is not equal to `address(0)`. `sDYSON::restake` could check that the additional `amount` staked is greater than 0.		
A5	Additional requirements in sDYSON::migrate	INFO
Function `sDYSON::migrate` could implement a require statement which ensures that `vault.dysonAmount > 0` and `vault.sDYSONAmount > 0`.		
A6	Additional requirement in AgentNFT::approve	INFO
An additional requirement, `require(to != address(msg.sender), "...")`, could be added to `AgentNFT::approve` to ensure that the `to` parameter (operator) is not equal to msg.sender.		
A7	transferOwnership functions could check that the new owner is different from address(0)	RESOLVED
The `transferOwnership` functions could add a requirement that ensures the new owner is different from address(0) in an attempt to prevent accidental mistakes.		
A8	Inaccurate docs comment	INFO
The Dyson documentation mentions that "after applying for withdrawal, the personal balance in the pool will down to zero immediately". This statement does not capture the partial withdrawal functionality offered by the `Gauge::applyWithdrawal` function.		
A9	sDYSON Stake and Unstake events do not mention the vault's index	INFO

The `Stake` and `Unstake` events emitted by `sDYSON::stake` and `sDYSON::unstake` respectively do not include the index of the vault that the user stakes into or unstakes from.		
A10	Gauge::applyWithdrawal could be renamed to requestWithdrawal	**INFO**
The function `Gauge::applyWithdrawal` can be renamed to `requestWithdrawal` to better describe its intent, which is to initiate/request the withdrawal of sDYSON from the Gauge contract.		
A11	sDYSON::currentModel might be uninitialized	**INFO**
The storage variable `currentModel` of the `sDYSON` contract, which can be set by the `setStakingRateModel` function, is not set during contract construction. If `currentModel` mistakenly remains uninitialized, calls to `getStakingRate` will fail unexpectedly, as it lacks a requirement that verifies that the variable has been set.		
A12	Compiler bugs	**INFO**
The codebase is compiled with version 0.8.17 of the Solidity compiler, which at the time of writing has no known bugs that can affect the correctness of the contracts. One should note that the aforementioned version is the latest version of the compiler and there might be bugs that have gone unnoticed so far.		

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.