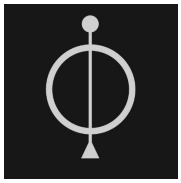

DYAD Stablecoin Audit Report

February 15th, 2023

Report Prepared By:
Zach Obront, Independent Security Researcher

Table of Contents

Table of Contents **1**

Summary **3**

Summary of Findings 3

Critical Severity Findings **4**

[C-01] User can steal vault funds by sequencing deposit, rebase, and redeem 4

[C-02] Can steal all dNFTs and funds from contract by flash loaning deposits & liquidations 7

[C-03] Using oracle provided price for deposits and redemptions allows user to steal funds 9

High Severity Findings **11**

[H-01] If insider deposits and unlocks in quick succession, attacker can steal their NFT and their deposit funds 11

[H-02] Current liquidation mechanism is ineffective and dangerous 12

[H-03] Collateralization ratio can be broken by users redeeming deposits for ETH 15

Medium Severity Findings **18**

[M-01] MIN_MINT_DYAD_DEPOSIT doesn't enforce any behavior on minters 18

[M-02] Check for stale data before trusting Chainlink's response 19

Low Severity & Informational Findings **20**

[L-01] All permissions are equivalent, so the PermissionsManager system isn't needed 20

[L-02] Missing checks for sending assets to non existent dNFTs 21

[L-03] Use SafeMint when minting new dNFTs 22

Summary

The DyadStablecoin/contracts-v3 repository was audited at the 02daa6d0accf7ae43e31113854a1771689f46a08 commit.

In scope were the following contracts:
- src/core/DNft.sol
- src/core/Dyad.sol
- src/core/PermissionManager.sol
- src/libraries/PermissionMath.sol

After completion of the fixes, the 227bf1618b79bd1efae1bd7f076c41cd100b3db7 commit was reviewed.

Summary of Findings

	Issues Found	Issues Resolved
Critical Severity	3	2
High Severity	3	3
Medium Severity	2	2
Informational	3	3

Critical Severity Findings

[C-01] User can steal vault funds by sequencing deposit, rebase, and redeem

When a user deposits into the protocol, their msg.value is converted into a number of shares in two steps:

- _eth2dyad() calculates the USD value of the ETH, based on the current oracle.
- _deposit2Shares() calculates the number of shares they should receive for that USD value.

The issue lies in the fact that the USD-ETH conversion used for Step 1 is out of sync with Step 2. Specifically, Step 1 relies on the current Chainlink Oracle, while Step 2 relies on the saved totalDeposit value to account for moving prices, which requires rebase() being called.

The result is that, if a user deposits before rebase() is called, they capture the benefits of the higher exchange rate for their ETH. And when rebase() is called, they capture an additional appreciation of their shares.

This is a significant problem for two reasons.

Problem 1: The adjustment doesn't just give them more upside relative to other users. It actually breaks the vault accounting and can leave the protocol insolvent. This is because the withdrawal process is based on the current price of ETH, and the user in the above situation is entitled to more ETH than is actually in the vault.

For a simple example, you can walk through the math imagining they are the first depositor:

- The initial price of ETH in the vault is 1000 USD/ETH
- The price increases to 1100 USD/ETH, but rebase hasn't been called yet
- Alice deposits 5 ETH, and receives 5500 shares as a result
- At this point, totalDeposits also equals 5500
- Rebase is called, and totalDeposits is increased by 10%, moving to 6050
- Alice tries to withdraw her 5500 shares = 6050 DYAD = 5.5 ETH
- The vault is insolvent, because there is only 5 ETH in it

Problem 2: More importantly, the appreciation from rebase() is a percentage, not a fixed value. The result is that, if the deposit() is very large (say, through a flash loan), a user is able to use the insolvency to withdraw the full holdings of the vault. See the Proof of Concept below for an example.

Proof of Concept

```
function test_CanLiquidateProtocolOnEthMove() public {
  uint id1 = dNft.mint{value: 100 ether}(address(this));

  address attacker = makeAddr("attacker");
  vm.deal(attacker, 5 ether);
  console.log("dNFT Vault Starting Balance: ", address(dNft).balance);
  console.log("Attacker Starting Balance: ", attacker.balance);
  vm.prank(attacker);
  uint id2 = dNft.mint{value: 5 ether}(attacker);

  oracleMock.setPrice(1100e8); // 10% increase

  // Borrow a flash loan and being the attack...
  vm.deal(attacker, 1_000_000 ether);
  vm.startPrank(attacker);
  dNft.deposit{value: 1_000_000 ether}(id2);

  // Rebase to adjust totalDeposit up, even though we already bought at the higher value.
  dNft.rebase();

  // Now redeem the full value of the vault (not all of our assets, because it would revert).
  dNft.redeemDeposit(id2, attacker, address(dNft).balance * 1100);

  // Return the flash loan...
  payable(address(0)).transfer(1_000_000 ether);

  console.log("dNFT Vault Ending Balance: ", address(dNft).balance);
  console.log("Attacker Ending Balance: ", attacker.balance);
}
```

```
Logs:
  dNFT Vault Starting Balance:  100000000000000000000
  Attacker Starting Balance:  5000000000000000000
  dNFT Vault Ending Balance:  0
  Attacker Ending Balance:  105000000000000000000
```

Recommendation

At the very least, all deposit, withdraw & redeem functions should use the ETH price saved in storage, and rebase() should be called at the beginning of each of those functions. This will ensure the two values are kept aligned for all transfers of value.

(Note: This also requires editing rebase() to ensure it simply exits early if an equal ETH price is returned, rather than reverting.)

However, that still leaves open the possibility that users can take advantage of price moves that haven't yet been reflected on the Chainlink Oracle to get a favorable price. I've seen a few different models for resolving this but all are complex: the best is probably to save "orders" and then process them at the next price from Chainlink, regardless of what it is.

Review

Timeout mechanism implemented in #28 and #29 with rebase modifier in #24 solves the immediate risk of this being exploitable to steal all the protocol funds.

Since the rebase modifier is used on all functions that access ETH price, the recommendation to pull ETH price from state is implemented in #31.

The larger concern around favorable prices is addressed in C-03.

[C-02] Can steal all dNFTs and funds from contract by flash loaning deposits & liquidations

When liquidate() is called for a given dNFT ID, there are two requirements:

- The current shares of that ID must be less than 0.1% of the total shares in the pool
- The shares after the liquidator's funds are deposited must by greater than 0.1% of the total shares in the pool

This can be exploited to steal any dNFT by depositing sufficient ETH to move the pool to greater than 1000x a given depositor, liquidating them and adding a small balance to push them over 0.1% of the pool, and withdrawing the initial deposit.

By sorting all dNFTs from lowest balance to highest, an attacker could perform this attack in such a way as to liquidate every dNFT, assuming sufficient funds (which would be easy to secure via Flash Loans).

Because internal balances are held by the dNFT rather than the user, this would also have the effect of moving the entire value of the vault to the attacker.

Proof of Concept

```
function test_CanLiquidateAnyone() public {
    address attacker = makeAddr("attacker");
    vm.deal(attacker, 6000 ether);
    address victim = makeAddr("victim");
    vm.deal(victim, 1000 ether);

    vm.prank(attacker);
    uint id1 = dNft.mint{value: 5 ether}(attacker);

    vm.prank(victim);
    uint id2 = dNft.mint{value: 5 ether}(victim);

    vm.startPrank(attacker);
    dNft.deposit{value: 5000 ether}(id1);
    dNft.liquidate{value: 1 ether}(id2, attacker);
    dNft.redeemDeposit(id1, attacker, 5000 ether);

    assertEq(dNft.ownerOf(id2), attacker);
}
```

Recommendation

The logic of the liquidation process needs to be rethought. Besides this possible attack, it seems that liquidating users when their shares fall below 0.1% of the total is likely to lead to unexpected dynamics and doesn't accomplish the goal of keeping the protocol solvent.

Review

The new liquidation mechanism (see fix for H-02) only liquidates if withdrawals exceed collateralization ratio, so this attack is no longer possible.

[C-03] Using oracle provided price for deposits and redemptions allows user to steal funds

New deposits use the _eth2dyad() function to calculate the exchange rate, which simplifies down to: AMOUNT_OF_ETH * ORACLE_PRICE_OF_ETH / 1e8;

Similarly, all redeeming uses _dyad2eth(), which calculates the exchange rate as: AMOUNT_OF_DYAD * 1e8 / ORACLE_PRICE_OF_ETH;

The problem is that a user can know which direction the oracle price of ETH will move before it does, either through watching the ETH price move in advance of the oracle updating, or watching the mempool for oracle update transactions and frontrunning them.

Note that this issue is distinct from C-01 because it exists even if rebase() is automatically called within each function and stays perfectly up to date.

Proof of Concept

There are two ways this knowledge can be used to attack the protocol.

1) A user can perform arbitrage within the protocol, dodging price updates that aren't in their favor, to take a larger proportion of shares than they should have:

- The pool is 20 ETH, all was bought at $1k, and I'm 50% of the pool.
    - eth2Dyad(20 ETH) = 20k
    - totalShares = 20k
    - totalDeposit = 20k
    - myShares = 10k
- I see that eth is moving to $995, so I frontrun and withdraw max (25%) into DYAD.
    - totalShares = 20k - 2.5k = 17.5k
    - totalDeposit = 20k - 2.5k = 17.5k
    - myShares = 7.5k
    - my DYAD: 2.5k
- When the oracle updates ETH price down to $995:
    - supplyDelta = 20k * 0.005 = 100
    - totalDeposit = 17,400
- I backrun the oracle update and immediately deposit all my DYAD back:
    - _depositToShares(2.5k) = 2514
    - myShares = 7.5k + 2514 = 10,014
    - totalShares = 17.5k + 2514 = 20,014
- The result is that I now own 50.03% of the pool and the other people own 49.97%. While these numbers may seem small, this attack can be repeated at each price update until I slowly dilute everyone else.

While this first attack can be stopped by implementing a 1 hour waiting period on all withdrawals, the second attack can be hedged out in order to ensure profitability, even with a forced waiting window.

2) A user can take advantage of the fact that depositing ETH into the pool is equivalent to buying slightly leveraged ETH, and can time their buys to earn extra on their leverage, hedging out the risk until they can withdraw.

- Let's say the pool leverage is 1.25x. There's 10 ETH in the pool (none of it mine). All ETH was bought at $1k, and then max DYAD was withdrawn:
    - totalShares = 10k - 2.5k withdrawal = 7.5k
    - totalDeposits = 10k - 2.5k withdrawal = 7.5k
- I see that ETH is moving to $1005 and I want to get a levered version of that 0.5% move, so I frontrun the deposit with another 10 ETH:
    - totalShares = 17.5k
    - totalDeposits = 17.5k
    - myShares = 10k
- When the rebase happens, totalDeposits goes up by 0.5% of DYAD + dD, which equals 100:
    - totalDeposits = 17,600
    - since i have 10k/17.5k shares, my share of the deposits is 10,057
    - at current ETH prices, this is 10.00696517 ETH, but i only put in 10 ETH
- In order to ensure a risk-free trade, at the same time I do this, I short 10 ETH elsewhere:
    - I'm now protected from ETH price movement risk
    - 1 hour later, I withdraw and close the short position, earning ~0.007 ETH profit
- Again, this seems small, but the numbers in the example were small. It seems reasonable that a malicious user could 1000x that, which would claim a 6.9 ETH profit each hour, at the expense of other protocol users.

Recommendation

Unfortunately this attack is foundational to the nature of pulling exchange rates directly from an oracle. To protect against this attack requires a foundational change in rearchitecture, for example, storing deposits and redemptions and executing them at a future exchange rate.

Review

The DYAD team is working on a new oracleless mechanism for ensuring sufficient collateral. The code will be reaudited when this change is complete.

High Severity Findings

[H-01] If insider deposits and unlocks in quick succession, attacker can steal their NFT and their deposit funds

The dNFT contract allows the owner to mint a predefined quantity of "insider" NFTs without any deposit attached to them. These NFTs begin in a locked state, which stops them from being immediately liquidated due to their lack of deposits.

The protocol enforces that, in order for insider's to mint any DYAD, they must unlock their NFTs (so that they will be subject to liquidation, like all other users).

However, there is no safety check for the opposite case, where an insider unlocks their NFT before making a deposit. In this situation, any user could liquidate them and steal their NFT.

This is especially dangerous because if a user calls both of these functions in quick succession, they may both be in the mempool at the same time. If this is the case, a malicious attacker can create a flashbots bundle to sandwich their liquidation transaction between the unlock() and deposit() transactions, with the result that:

- The attacker will successfully liquidate and steal the insider's NFT
- The deposit transaction will deposit the insider's ETH to the stolen NFT, securing it for the attacker

Recommendation

I would recommend adding a check to the unlock() function to ensure this situation is avoided:

```
function unlock(uint id)
external
    isNftOwner(id)
{
    if (!id2Locked[id]) revert NotLocked();
    if (id2Shared[id] == 0) revert MustDepositFirst();
    id2Locked[id] = false;
    emit Unlocked(id);
}
```

Note: This requires adding a MustDepositFirst() error to IDNft.sol.

Review

The new liquidation mechanism (see fix for H-02) only liquidates if withdrawals exceed collateralization ratio, so this attack is no longer possible.

[H-02] Current liquidation mechanism is ineffective and dangerous

The liquidation mechanism is intended to keep the protocol overcollateralized. However, the current mechanism design seems inherently unstable and doesn't accomplish the objective.

Currently, the logic for liquidation is as follows:

- The shares held by a specific dNFT falls below a certain threshold (currently 0.1%)
- A Liquidator sends sufficient ETH to push that dNFT's shares above the threshold
- The Liquidator gets possession of the dNFT, as well as all the shares that it previously owned
- There are a number of issues I see with this. I'll go into details below, and then share some thoughts on how this might look in the Recommendations section.

No Stable Equilibrium

There is no game theoretically stable place for this system to land. The specifics will depend on the threshold that is chosen, but at most thresholds there will always be users who are near the line (at some, like the current 0.1%, it is a mathematical guarantee that at least half of the users will be liquidatable at all times).

This creates a competitive dynamic that users will always need to be fighting to be in the top 50%, liquidating other users, and depositing ETH in to stay near the top.

While this is reminiscent of some of the most fun degen projects of 2022, the goal for core is to provide stability, and these dynamics are not conducive to such stability.

Not Fair to Users Who Get Liquidated

Even in the even that we want these competitive dynamics, there are a spectrum of possibilities between favoring the user who is liquidated or who is the liquidator.

This system heavily favors the liquidator, in a way that feels unfair to normal users.

If a user's number of shares falls even 1% shy of the threshold, a liquidator can add that 1% and take the original user's 99%, as well as their dNFT. A more reasonable system might require the liquidator to put up the full 100%, and refund the original user their 99%, losing them their dNFT only.

Doesn't Accomplish the Stated Goal

We need to use liquidations as a tool to keep the system overcollateralized. This means, as much as possible, we should (a) understand the situations that might cause

undercollateralization and allow liquidations to resolve this issue, and (b) not have liquidations impact behavior when they don't support this goal.

Looking at a few examples, we can see that the current example is off:

- A user with 1% of the dD but 20% of the withdrawn DYAD is contributing massively to missing the collateralization ratio, but would not be liquidatable
- A user with 0.01% of the dD and no withdrawn DYAD is not negatively impacting the collateralization ratio, but would be liquidatable

Easily Abusable

Since users who might be performing the liquidations also have the ability to impact the totalDeposits (which is the denominator in calculating the liquidation threshold), it's easy for a user to put the contract in a state that allows liquidations that wouldn't otherwise be possible. This is similar to issue C-02.

Recommendations

If our goals are to have liquidations keep the protocol overcollateralized, the simplest way to accomplish this is to force every dNFT to be similarly overcollateralized.

This also has the advantage of keeping liquidations based on a user's own decisions, rather than outside circumstances that can be manipulated by other users.

Furthermore, enforcing collateralization on the user level avoids the unfair situation that is currently possible, where one user gets more "benefit" in the form of withdrawing DYAD than another, because the first user withdraws first, and the second is therefore unable to because of the system-wide collateralization ratio.

This is difficult to accomplish, especially in the current system where the ideal invariant of totalDeposit * ETH price == address(this).balance is not enforced. However, if changes are made to the rebasing process to ensure that this becomes a true invariant, it opens up the ability to track this properly.

One way to accomplish this would be:

- We can create a _shares2Deposit() function that calculates the underlying DYAD value of a number of shares
- We can add a mapping (uint => uint) dyadWithdrawn that tracks the DYAD that's been withdrawn for each dNFT
- When redeeming (ETH) or withdrawing (DYAD), we can check the user invariant that, after the changes are made, _shares2Deposit(id2Shares[id]).divWadDown(dyadWithdrawn[id]) >= MIN_COLLATERIZATION_RATIO.
- There could still be a situation where a user falls below the MIN_COLLATERIZATION_RATIO based on the price of ETH falling. In this case, they

would be susceptible to a gentle liquidation to incentivize a third party to bring the dNFT back into the correct range (ie the user would have their ETH refunded and the liquidator would supply the replacement ETH, and would get the dNFT as a reward).

Review

Fix confirmed in #20 and #23.

[H-03] Collateralization ratio can be broken by users redeeming deposits for ETH

A key property of the DYAD system is that the collateralization ratio (300%) is maintained. This means that for every 1 DYAD in circulation, there is 3x as much ETH (priced in USD) in the vault.

This invariant is enforced in the withdraw() function, which stops users from minting more DYAD when such a mint would break the invariant:

```
function withdraw(uint from, address to, uint amount) external
    isNftOwnerOrHasPermission(from, Permission.WITHDRAW)
    isUnlocked(from)
{
    _subDeposit(from, amount);

    uint collatVault    = address(this).balance * _getEthPrice()/1e8;
    uint newCollatRatio = collatVault.divWadDown(dyad.totalSupply() + amount);
    if (newCollatRatio < MIN_COLLATERIZATION_RATIO) { revert CrTooLow(); }
    ...
}
```

However, the same check is not enforced when redeeming ETH out of the contract. Since a key goal is keeping the ratio of circulating DYAD and ETH bounded by this ratio, it is crucial that we enforce this check on both DYAD minting and ETH redeeming.

Proof of Concept

Here is a test showing that we can get the collateralization ratio as low as 1:1 by withdrawing all non-minted deposits:

```
function test_CollateralizationRatioBrokenOnRedeemDeposit() public {
    // We deposit 5000 in totalDeposit and mint 1000 of them. Ratio is $5000 of ETH / 1000
supply.
    uint id1 = dNft.mint{value: 5 ether}(address(this));
    dNft.withdraw(id1, address(this), 1000e18);
    console.log(_calculateCollateralizationRatio()); // returns 5e18 - success

    // We can now withdraw all the remaining ETH with no check.
    dNft.redeemDeposit(id1, address(1), 4000e18);
    console.log(_calculateCollateralizationRatio()); // returns 1e18 - uh oh
}
```

```
function _calculateCollateralizationRatio() internal returns(uint) {
    uint ETH_PRICE = 1000 * 1e8;
    uint MIN_COLLATERIZATION_RATIO = 3e18;
    uint collatVault = address(dNft).balance * ETH_PRICE/1e8;
    uint newCollatRatio = FixedPointMathLib.divWadDown(collatVault, dyad.totalSupply());
    return newCollatRatio;
}
```

Recommendation

I would recommend moving the collateralization logic into a modifier, as follows:

```
modifier collateralizationCheck(uint _amountMinted, uint _amountRedeemed) {
    uint collatVault = (address(this).balance - _dyad2eth(_amountRedeemed)) *
_getEthPrice() / 1e8;
    if (dyad.totalSupply() > 0) {
        uint newCollatRatio = collatVault.divWadDown(dyad.totalSupply() + _amountMinted);
        if (newCollatRatio < MIN_COLLATERIZATION_RATIO) { revert CrTooLow(); }
    }
    _;
}
```

This modifier could then be implemented by both the functions listed below:

```
function withdraw(uint from, address to, uint amount) external
    isNftOwnerOrHasPermission(from, Permission.WITHDRAW)
    isUnlocked(from)
    collateralizationCheck(amount, 0)
{ ... }
```

and

```
function redeemDeposit(uint from, address to, uint amount) external
    isNftOwnerOrHasPermission(from, Permission.REDEEM)
    isUnlocked(from)
    collateralizationCheck(0, amount)
    returns (uint) { ... }
```

You'll note a few small additional changes:

- We need to check whether dyad.totalSupply() == 0 before performing this logic, because the collateralization ratio is infinite before any DYAD has been minted, and we will revert when dividing by zero in our check. This is included in the modifier above.

- Your existing testCannot_WithdrawCrTooLow test is broken by this change, but it appears this test is incorrect. It is expecting a revert when a dNFT is minted and all is withdrawn, but this situation should be fine, as it does not break any collateralization ratio is there is no DYAD yet in existence.

Review

Fix confirmed in #20 by including a mechanism to check individual user collateralization ratios on withdrawals.

Medium Severity Findings

[M-01] MIN_MINT_DYAD_DEPOSIT doesn't enforce any behavior on minters

The public mint() function includes a requirement that users must submit $5k of ETH in order to mint a new dNFT. However, there is no restriction stopping them from immediately redeeming this deposit back into ETH by calling redeemDeposit().

The only requirement is that they keep sufficient balance to avoid being liquidated, which is enforced independently from the MIN_MINT_DYAD_DEPOSIT.

The result is that a user can buy up the full supply of dNFTs for close to free, as they loop through:

- mint() => deposit $5k of ETH
- redeemDeposit() => cash out up to $5k of ETH

This becomes a greater risk depending how the liquidation() function is changed, based on #4.

Recommendations

This relies heavily on how you decide to implement liquidations, as it's unclear whether they will solve for this use case.

In the event that liquidations do not solve this problem (which is my recommendation), consider requiring a minimum number of shares (or value of deposits) that must be maintained by all dNFTs at all times. This will ensure that a fixed amount of the deposited dollars stay in the vault.

Review

Fix confirmed in #25, 881163c, #30.

The deposit minimum is still not a true invariant, as a user with minimum deposits could have their deposits lowered by the price of ETH falling. However, the goal of this check is to keep users from disengaging from the system, and there is no harm done or liquidation risk exists from allowing it to fall below the minimum, so this sufficiently accomplishes the goal.

[M-02] Check for stale data before trusting Chainlink's response

Much of the math in the protocol is based on the data provided by Chainlink's ETH-USD feed.

According to Chainlink's documentation, it is important to provide additional checks that the data is fresh:

- If answeredInRound is less than roundId, the answer is being carried over.
- A timestamp with zero value means the round is not complete and should not be used.

Recommendation

Add the following checks to the _getEthPrice() function to ensure the data is fresh and accurate:

```
function _getEthPrice() public view returns (uint) {
  (uint80 roundID, int256 price,, uint256 timeStamp, uint80 answeredInRound) =
oracle.latestRoundData();
  require(timeStamp != 0);
  require(answeredInRound >= roundID);
  return price.toUint256();
}
```

Review

Fix confirmed in #10.

Low Severity & Informational Findings

[L-01] All permissions are equivalent, so the PermissionsManager system isn't needed

The PermissionsManager enforces three types of permissions: MOVE, WITHDRAW, and REDEEM.

However, there is no need for such a system, because all three permissions are equivalent:

- A user with MOVE permissions can move shares to another account, and then withdraw or redeem
- A user with WITHDRAW permissions can withdraw DYAD tokens, transfer them, and either redeposit them into another account (move) or redeem them
- A user with REDEEM permissions can withdraw ETH, and use it to deposit and withdraw in a new account

Recommendation

Replace the PermissionsManager system with a simple approved boolean. We would still need the lastUpdated logic to ensure that old approvals are maintained as dNFTs are transferred.

Review

Fix confirmed in #12.

[L-02] Missing checks for sending assets to non existent dNFTs

There are two places in the code where the user inputs the id of the dNFT they would like to send assets to.

- In deposit(), the user sends msg.value directly to the dNFT
- In move(), the user sends their own shares to another dNFT

In both cases, there are no checks to ensure that the receiving dNFT actually exists.

This could allow users to deposit assets to a non-existent ID that would become permanently trapped in the contract.

Proof of Concept

```
function test_CanDepositToNonExistantId() public {
    dNft.deposit{value: 5000 ether}(987654321);
}
```

Recommendation

Add a check in both deposit() and move() to ensure that the ID exists:

```
function deposit(uint id) external payable returns (uint) {
    require(id < totalSupply(), "dNFT does not exist");
    return _deposit(id);
 }
```

Review

Fix confirmed by #11.

[L-03] Use SafeMint when minting new dNFTs

Both mint() functions to mint new dNFTs have a to field to input who will receive the NFT. This can be dangerous in the event that a user specifies a contract address that is not set up to handle NFTs, and the dNFT gets stuck.

This is harmful to the user, but also to the protocol. There are a limited number of dNFTs, and there's an expectation that they will be used to provide liquidity and add to the circulating supply of DYAD. Locked dNFTs are a drag on the system.

While the current liquidation system would give the ability to take the NFTs from these inactive accounts, the proposed new liquidation system (see #15) does not, so it is important that we do what we can to protect against these errors.

Recommendations

Use the _safeMint() function (already included in the imported OpenZeppelin ERC721 contract) instead of _mint().

Review

Fix confirmed in #18.