DYAD Security Review

Pashov Audit Group

Conducted by: Juan, DadeKuma, pontifex, ZanyBonzy

September 14th - September 17th

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **DyadStablecoin/contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About DYAD

The DYAD protocol enables users to mint interest-free stablecoins by depositing collateral, such as ETH, with a minimum collateral ratio as low as 100%. Users must own a DYAD-specific NFT, called a Note, to participate in the ecosystem, which tracks user activity and facilitates minting, yield farming, and transferring collateralized positions through share tokens of ERC-4626 vaults.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 973cb961198890449e0a80b4be4065dccff0abc0

fixes review commit hash - fe10994f7ba9477ca464c5ec286f99eaea1dc18c

Scope

The following smart contracts were in scope of the audit:

- `VaultManagerV5`
- `DyadXPv2`
- `UniswapV3Staking`

7. Executive Summary

Over the course of the security review, Juan, DadeKuma, pontifex, ZanyBonzy engaged with DYAD to review DYAD. In this period of time a total of **15** issues were uncovered.

Protocol Summary

Protocol Name	DYAD
Repository	https://github.com/DyadStablecoin/contracts
Date	September 14th - September 17th
Protocol Type	Stablecoin

Findings Count

Severity	**Amount**
Critical	2
High	6
Medium	3
Low	4
Total Findings	**15**

Summary of Findings

ID	Title	Severity	Status
[C-01]	The entire kerosene supply can be drained by claiming rewards	Critical	Resolved
[C-02]	Underlying tokens of UniV3 Liquidity Position are unchecked	Critical	Resolved
[H-01]	Users will get 1/100 of the intended accrual rate	High	Resolved
[H-02]	totalXP is wrongly calculated for existing users	High	Resolved
[H-03]	updateXP can be omitted during liquidation	High	Resolved
[H-04]	Initialization of DyadXPv2 is impossible	High	Resolved
[H-05]	Users cannot stake in UniV3Staking	High	Resolved
[H-06]	Staking rewards should be claimed before each balanceOfNote changing	High	Acknowledged
[M-01]	Users will miss some rewards when they add liquidity	Medium	Resolved
[M-02]	Reward time is updated even when no rewards are sent	Medium	Resolved
[M-03]	Liquidated users are wrongly protected from slashing due to incorrect XP update	Medium	Resolved
[L-01]	Owner is ignored during initialization	Low	Resolved
[L-02]	DOS of user operations if totalXP is below 10	Low	Resolved

[L-03]	Unused isDNftOwner can be removed	Low	Resolved
[L-04]	The unstake() function can be re-entered	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] The entire kerosene supply can be drained by claiming rewards

Severity

Impact: High

Likelihood: High

Description

This is the current formula used to calculate rewards when a note is staked:

```
function _calculateRewards(
   uint256noteId,
   StakeInfostoragestakeInfo
 ) internal view returns (uint256
            uint256 timeDiff = block.timestamp - stakeInfo.lastRewardTime;

            uint256 xp = dyadXP.balanceOfNote(noteId);

->          return timeDiff * rewardsRate * stakeInfo.liquidity * xp;
        }
```

The issue is that rewards can be extremely high, as the XP values are really big.

Kerosene has a total supply of: `1_000_000_000 1e18`, but most notes have too much XP.

In the best-case scenario, the XP will be larger than the total supply and it won't be possible to stake anything.

In the worst-case scenario, a user can claim most of the total supply with a minimal liquidity investment, as small as 1 wei.

Recommendations

The XP must be scaled down in the reward calculation. It is recommended to divide by `1e18` after multiplying `rewardsRate` and `stakeInfo.liquidity` together. Depending on the order of magnitude of `xp` values, the same will have to be done when multiplying by `xp`.

[C-02] Underlying tokens of UniV3 Liquidity Position are unchecked

Severity

Impact: High

Likelihood: High

Description

Within `UniswapV3Staking.stake()`, the underlying tokens of the staked UniV3 Liquidity Position are unchecked. This lets a malicious actor create a new UniV3 pool with their own arbitrary ERC20 tokens, where they mint themselves a large number of tokens. This causes the `liquidity` of their position in this pool to be extremely high, allowing them to earn an extremely large amount of rewards from staking this position.

Recommendations

Within `stake()`, ensure that the underlying tokens of the position are USDC and DYAD

8.2. High Findings

[H-01] Users will get 1/100 of the intended accrual rate

Severity

Impact: Medium

Likelihood: High

Description

The base accrual rate, as described in the documentation, should be `1e9`.

However, in `DyadXPv2._computeXP` it is set to `1e7`:

```
uint256 adjustedAccrualRate = accrualRateModifier * 1e7;
```

As such, users will get a meager yield for their deposit (2 orders of magnitude less than the intended value).

Recommendations

Consider the following change:

```
-    uint256 adjustedAccrualRate = accrualRateModifier * 1e7;
+    uint256 adjustedAccrualRate = accrualRateModifier * 1e9;
```

[H-02] `totalXP` is wrongly calculated for existing users

Severity

Impact: Medium

Likelihood: High

Description

In `src/staking/DyadXPv2.sol`, the contract must be initialized through the `initialize` function, and the `totalXP` of each user is also calculated here:

```
function initialize(address owner) public reinitializer(2) {
        __UUPSUpgradeable_init();
        __Ownable_init(msg.sender);

        uint256 dnftSupply = DNFT.totalSupply();

        for (uint256 i = 0; i < dnftSupply; ++i) {
            noteData[i] = NoteXPData({
                lastAction: uint40(block.timestamp),
                keroseneDeposited: uint96(KEROSENE_VAULT.id2asset(i)),
                lastXP: noteData[i].lastXP,
->              totalXP: noteData[i].lastXP,
                dyadMinted: DYAD.mintedDyad(i)
            });
        }
    }
```

However, the `totalXP` doesn't consider the last accrued bonus, as it is only the `lastXP` rather than the actual total (including the bonus) during the upgrade.

This is because the `elapsed` time between the upgrade and their last action is very likely to not be zero:

```
function _computeXP(NoteXPData memory lastUpdate) internal view returns
      (uint256) {
->      uint256 elapsed = block.timestamp - lastUpdate.lastAction;
        uint256 deposited = lastUpdate.keroseneDeposited;
        uint256 dyadMinted = lastUpdate.dyadMinted;
        uint256 totalXP = lastUpdate.totalXP;

        uint256 accrualRateModifier = totalXP > 0 ? 1e18 / totalXP.log10
          () : 1e18;

        uint256 adjustedAccrualRate = accrualRateModifier * 1e7;

        // bonus = deposited + deposited * (dyadMinted /
        //(dyadMinted + deposited))
        uint256 bonus = deposited;

        if (dyadMinted + deposited != 0) {
            bonus += deposited.mulWadDown(dyadMinted.divWadDown
              (dyadMinted + deposited));
        }

        return uint256(lastUpdate.lastXP +
          (elapsed * adjustedAccrualRate * bonus) / 1e18);
    }
```

A numerical example:

```
lastXP: 1e18

_computeXP:

elapsed: 10_000
deposited: 1e18
dyadMinted: 0
totalXP: 0

accrualRateModifier: 1e18

adjustedAccrualRate: 1e18 * 1e7 = 1e25

bonus: 1e18

0 + 1e18 != 0
    bonus = 1e18 + (1e18 * (0 / (0 + 1e18)) = 1e18

-> 1e18 + (10_000 * 1e25 * bonus) / 1e18 = 1e18 + 10_000 * 1e25
```

In this example, the user will have a `totalXP = 1e18` with the current code, but they should have `totalXP = 1e18 + 1e29` instead.

This will affect the `accrualRateModifier` the next time the function is called (for example, to calculate rewards of a Uniswap position), as the `totalXP` will be lower than intended.

Recommendations

Consider applying the following fix:

```
function initialize(address owner) public reinitializer(2) {
        __UUPSUpgradeable_init();
        __Ownable_init(msg.sender);

        uint256 dnftSupply = DNFT.totalSupply();

        for (uint256 i = 0; i < dnftSupply; ++i) {
+       uint256 totalXP = _computeXP(noteData[i]);
            noteData[i] = NoteXPData({
                lastAction: uint40(block.timestamp),
                keroseneDeposited: uint96(KEROSENE_VAULT.id2asset(i)),
                lastXP: noteData[i].lastXP,
-               totalXP: noteData[i].lastXP,
+               totalXP: totalXP,
                dyadMinted: DYAD.mintedDyad(i)
            });
        }
    }
```

[H-03] `updateXP` can be omitted during liquidation

Severity

Impact: Medium

Likelihood: High

Description

Though the `liquidate` function invokes `dyad.burn` the `updateXP` can be omitted in case the liquidated note has no `KEROSENE_VAULT` or `depositAmount == 0`. This way liquidated note's `balanceOfNote` will be incorrect.

```
function liquidate(uint256 id, uint256 to, uint256 amount)
        external
        isValidDNft(id)
        isValidDNft(to)
        returns (address[] memory, uint256[] memory)
    {
        uint256 cr = collatRatio(id);
        if (cr >= MIN_COLLAT_RATIO) revert CrTooHigh();
        uint256 debt = dyad.mintedDyad(id);
>>      dyad.burn(id, msg.sender, amount); // changes `debt` and `cr`

        lastDeposit[to] = block.number; // `move` acts like a deposit

        uint256 numberOfVaults = vaults[id].length();
        address[] memory vaultAddresses = new address[](numberOfVaults);
        uint256[] memory vaultAmounts = new uint256[](numberOfVaults);

        uint256 totalValue = getTotalValue(id);
        if (totalValue == 0) return (vaultAddresses, vaultAmounts);

        for (uint256 i = 0; i < numberOfVaults; i++) {
            Vault vault = Vault(vaults[id].at(i));
            vaultAddresses[i] = address(vault);
            if (vaultLicenser.isLicensed(address(vault))) {
                uint256 depositAmount = vault.id2asset(id);
>>              if (depositAmount == 0) continue;
                uint256 value = vault.getUsdValue(id);
                uint256 asset;
                if (cr < LIQUIDATION_REWARD + 1e18 && debt != amount) {
                    uint256 cappedCr = cr < 1e18 ? 1e18 : cr;
                    uint256 liquidationEquityShare =
                      (cappedCr - 1e18).mulWadDown(LIQUIDATION_REWARD);
                    uint256 liquidationAssetShare =
                      (liquidationEquityShare + 1e18).divWadDown(cappedCr);
                    uint256 allAsset = depositAmount.mulWadUp
                      (liquidationAssetShare);
                    asset = allAsset.mulWadDown(amount).divWadDown(debt);
                } else {
                    uint256 share = value.divWadDown(totalValue);
                    uint256 amountShare = share.mulWadUp(amount);
                    uint256 reward_rate = amount.divWadDown(debt).mulWadDown
                      (LIQUIDATION_REWARD);
                    uint256 valueToMove = amountShare + amountShare.mulWadUp
                      (reward_rate);

                                        uint256 cappedValue = valueToMove > value ? v
                    asset = cappedValue * (10 ** (vault.oracle().decimals
                      () + vault.asset().decimals()))
                        / vault.assetPrice() / 1e18;
                }
                vaultAmounts[i] = asset;

                vault.move(id, to, asset);
>>              if (address(vault) == KEROSENE_VAULT) {
                    dyadXP.updateXP(id);
                    dyadXP.updateXP(to);
                }
            }
        }

        emit Liquidate(id, msg.sender, to, amount);

        return (vaultAddresses, vaultAmounts);
    }
```

Recommendations

Consider updating XP of the liquidated note after the debt changes.

[H-04] Initialization of DyadXPv2 is impossible

Severity

Impact: Medium

Likelihood: High

Description

`DyadXPv2.initialize()` loops across the supply of `DNFT`:

```
for (uint256 i = 0; i < dnftSupply; ++i) {
            noteData[i] = NoteXPData({
                lastAction: uint40(block.timestamp),
                keroseneDeposited: uint96(KEROSENE_VAULT.id2asset(i)),
                lastXP: noteData[i].lastXP,
                totalXP: noteData[i].lastXP,
                dyadMinted: DYAD.mintedDyad(i)
            });
        }
```

The current total supply of `DNFT` is 882.

gas used in 88 iterations is 3.130682e6

This means that the total cost of the loop is at least 3.130682e7, which exceeds the 3e7 (30M) gas limit.

Proof of Concept

Due to foundry limitations, it's not possible to run the PoC across all 882 iterations.

To calculate the gas used for 88 (10% of total) iterations, make the following change in `DyadXPv2.initialize()`:

```
noteData[i] = NoteXPData({
                lastAction: uint40(block.timestamp),
                keroseneDeposited: uint96(KEROSENE_VAULT.id2asset(i)),
                lastXP: noteData[i].lastXP,
                totalXP: noteData[i].lastXP,
                dyadMinted: DYAD.mintedDyad(i)
            });

+         if (i == dnftSupply / 10 - 1) {
+              console.log
+ ("gas used in %s iterations is %e", i+1, gas - gasleft());
+                return;
+           }
```

Then add this PoC file to `test/` and run `test_InitCost`:

Coded PoC

```
import {Test, console} from "forge-std/Test.sol";
import {DyadXPv2} from "../../src/staking/DyadXPv2.sol";

import {Kerosine} from "../../src/staking/Kerosine.sol";
import {DNft} from "../../src/core/DNft.sol";
import {KeroseneVault} from "../../src/core/VaultKerosene.sol";
import {Dyad} from "../../src/core/Dyad.sol";
import
   {UUPSUpgradeable} from "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgrade

contract StakingTest is Test {
    address VAULT_MANAGER = 0xB62bdb1A6AC97A9B70957DD35357311e8859f0d7;
    address dNFT_owner = 0xDeD796De6a14E255487191963dEe436c45995813;

    DyadXPv2 dyadXP_v2;

    Kerosine kerosine;
    DNft dnft;
    Dyad dyad;
    KeroseneVault keroseneVault;

    function setUp() external {
        string memory RPC_URL =
        // "https://eth-mainnet.g.alchemy.com/v2/vDqr_aMYwqkkAUs-sM3S07j92pkny3yE";
        vm.createSelectFork(RPC_URL);

        dyad = Dyad(0xFd03723a9A3AbE0562451496a9a394D2C4bad4ab);
        dnft = DNft(0xDc400bBe0B8B79C07A962EA99a642F5819e3b712);
        kerosine = Kerosine(0xf3768D6e78E65FC64b8F12ffc824452130BD5394);
        keroseneVault = KeroseneVault
          (0x4808e4CC6a2Ba764778A0351E1Be198494aF0b43);
    }

    function test_InitCost() public {
        // Deploy DyadXPv2 implementation
        DyadXPv2 impl = new DyadXPv2(
            VAULT_MANAGER,
            address(keroseneVault),
            address(dnft),
            address(dyad)
        );

        address xp_owner = 0xDeD796De6a14E255487191963dEe436c45995813;

        // Upgrade DyadXP->DyadXPv2
        vm.prank(xp_owner);
        UUPSUpgradeable xp_proxy = UUPSUpgradeable(payable
          (0xeF443646E52d1C28bd757F570D18F4Db30dB70F4));
        xp_proxy.upgradeToAndCall(address(impl), abi.encodeWithSignature
          ("initialize(address)", address(this)));
        dyadXP_v2 = DyadXPv2(address(xp_proxy));
    }
}
```

Console output:

```
Ran 1 test for test/CantInit.t.sol:StakingTest
[PASS] test_InitCost() (gas: 4767974)
Logs:
  gas used in 89 iterations is 3.18022e6
```

Recommendations

Consider not looping across the entire dnftSupply on initialization.

[H-05] Users cannot stake in `UniV3Staking`

Severity

Impact: Medium

Likelihood: High

Description

`UniswapV3Staking.stake()` pulls NFTs from the user in the following way:

```
positionManager.safeTransferFrom(msg.sender, address(this), tokenId);
```

However, this will end up calling the `onERC721Received()` function on the `UniswapV3Staking` contract, and expect a return value. However since that function is not implemented in this contract, the ERC721 transfer will fail, reverting the execution of `stake()`.

Recommendations

Implement the following function to ensure that ERC721's can be received via `safeTransferFrom()`:

```
function onERC721Received
  (address, address, uint256, bytes calldata) public pure returns (bytes4) {
        return msg.sig;
}
```

[H-06] Staking rewards should be claimed before each `balanceOfNote` changing

Severity

Impact: Medium

Likelihood: High

Description

Since staking rewards depend on `balanceOfNote` the `claimRewards` function should be invoked on each `balanceOfNote` changing. But this functionality is not implemented. This way reward can be calculated incorrectly which can cause sufficient asset losses.

```
function _calculateRewards(
      uint256noteId,
      StakeInfostoragestakeInfo
    ) internal view returns (uint256
        uint256 timeDiff = block.timestamp - stakeInfo.lastRewardTime;

>>      uint256 xp = dyadXP.balanceOfNote(noteId);

>>      return timeDiff * rewardsRate * stakeInfo.liquidity * xp;
    }
```

Recommendations

Consider claiming rewards for notes with `stakes[noteId].isStaked == true` before any `balanceOfNote` changing: `deposit`, `withdraw`, `mintDyad`, `burnDyad`, `liquidate` (for both `id` and `to` notes).

8.3. Medium Findings

[M-01] Users will miss some rewards when they add liquidity

Severity

Impact: Medium

Likelihood: Medium

Description

In `UniswapV3Staking.stake`, the current liquidity is stored when a note is staked:

```
function stake(uint256 noteId, uint256 tokenId) external {
        require(dnft.ownerOf
          (noteId) == msg.sender, "You are not the Note owner");

        StakeInfo storage stakeInfo = stakes[noteId];
        require(!stakeInfo.isStaked, "Note already used for staking");

        (,,,,,,, uint128 liquidity,,,,) = positionManager.positions(tokenId);
        require(liquidity > 0, "No liquidity");

        positionManager.safeTransferFrom(msg.sender, address(this), tokenId);

        stakes[noteId] = StakeInfo({
->        liquidity: liquidity,
          lastRewardTime: block.timestamp,
          tokenId: tokenId,
          isStaked: true
        });

        emit Staked(msg.sender, noteId, tokenId, liquidity);
    }
```

This value will be used when they claim their rewards:

```
function _calculateRewards(
      uint256noteId,
      StakeInfostoragestakeInfo
    ) internal view returns (uint256
        uint256 timeDiff = block.timestamp - stakeInfo.lastRewardTime;

        uint256 xp = dyadXP.balanceOfNote(noteId);

->      return timeDiff * rewardsRate * stakeInfo.liquidity * xp;
    }
```

However, if a user keeps adding liquidity to their own position with `NonfungiblePositionManager.increaseLiquidity`, the liquidity will be stale, so they will receive fewer rewards than intended as it will use the original value (when they staked it).

Recommendations

Consider using the up-to-date liquidity instead of storing it:

```
function _calculateRewards(
      uint256noteId,
      StakeInfostoragestakeInfo
    ) internal view returns (uint256
        uint256 timeDiff = block.timestamp - stakeInfo.lastRewardTime;

        uint256 xp = dyadXP.balanceOfNote(noteId);

-       return timeDiff * rewardsRate * stakeInfo.liquidity * xp;
+       (,,,,,,, uint128 liquidity,,,,) = positionManager.positions
+ (stakeInfo.tokenId);
+       return timeDiff * rewardsRate * liquidity * xp;
    }
```

[M-02] Reward time is updated even when no rewards are sent

Severity

Impact: Medium

Likelihood: Medium

Description

In `UniswapV3Staking` users can claim rewards by calling the `claimRewards` function. However, the `lastRewardTime` is always updated even when they get

zero rewards:

```
function _claimRewards(
      uint256noteId,
      StakeInfostoragestakeInfo,
      addressrecipient
    ) internal {
        require(dnft.ownerOf
          (noteId) == msg.sender, "You are not the Note owner");
        require(stakeInfo.isStaked, "Note not staked");
        uint256 rewards = _calculateRewards(noteId, stakeInfo);
->      stakeInfo.lastRewardTime = block.timestamp;

        if (rewards > 0) {
            rewardsToken.transferFrom(rewardsTokenHolder, recipient, rewards);
            emit RewardClaimed(recipient, rewards);
        }
    }
```

Suppose the scenario where rewards are temporarily disabled by setting `rewardsRate` to zero. Then, any user that calls `claimRewards` during this period will update their `lastRewardTime` even if they don't get anything back. The next time, they will receive fewer rewards after they are enabled again.

Recommendations

Consider applying the following change:

```
function _claimRewards(
      uint256noteId,
      StakeInfostoragestakeInfo,
      addressrecipient
    ) internal {
        require(dnft.ownerOf
          (noteId) == msg.sender, "You are not the Note owner");
        require(stakeInfo.isStaked, "Note not staked");
        uint256 rewards = _calculateRewards(noteId, stakeInfo);
-       stakeInfo.lastRewardTime = block.timestamp;

        if (rewards > 0) {
+           stakeInfo.lastRewardTime = block.timestamp;
            rewardsToken.transferFrom(rewardsTokenHolder, recipient, rewards);
            emit RewardClaimed(recipient, rewards);
        }
    }
```

[M-03] Liquidated users are wrongly protected from slashing due to incorrect XP update

Severity

Impact: Medium

Likelihood: Medium

Description

On a base level, liquidating a vault functions as both a `withdraw` and `deposit` function and upon withdrawing from a kerosene vault, the note's xp is slashed. However, during liquidations, this is not done for the liquidated note, as `updateXP` is used instead, rather than the `beforeKeroseneWithdrawn`. As a result, the note xp is not affected even during the pseudowithdrawal.

```
vault.move(id, to, asset);
                if (address(vault) == KEROSENE_VAULT) {
                    dyadXP.updateXP(id);
                    dyadXP.updateXP(to);
                }
            }
```

Recommendations

Fixing this might need a bit of refactoring. Something like below will work.

```
vaultAmounts[i] = asset;

-                vault.move(id, to, asset);
-                if (address(vault) == KEROSENE_VAULT) {
-                    dyadXP.updateXP(id);
-                    dyadXP.updateXP(to);
-                }
-            }
+              if (address(vault) == KEROSENE_VAULT) {
+                  dyadXP.beforeKeroseneWithdrawn(id);
+                  vault.move
+ (id, to, asset);//This subtracts from id first and adds to to
+                  dyadXP.updateXP(to);
+               } else {
+               vault.move(id, to, asset);
+           }
```

8.4. Low Findings

[L-01] Owner is ignored during initialization

In `DyadXPv2` and `UniswapV3Staking` the `initialize` function has an `owner` argument; However, this argument is ignored during the ownable initialization, as the `msg.sender` is set as the owner instead:

```
function initialize(
->      address _owner,  //@audit ignored
        IERC20 _rewardsToken,
        INonfungiblePositionManager _positionManager,
        IDyadXP _dyadXP,
        DNft _dnft,
        uint256 _rewardsRate,
        address _rewardsTokenHolder
    ) public initializer {
        __UUPSUpgradeable_init();
->      __Ownable_init(msg.sender);

        rewardsToken = _rewardsToken;
        positionManager = _positionManager;
        dyadXP = _dyadXP;
        dnft = _dnft;
        rewardsRate = _rewardsRate;
        rewardsTokenHolder = _rewardsTokenHolder;
    }
```

This might result in the wrong owner being set when `msg.sender` differs from `_owner`. Either remove the `owner` argument from `initialize`, or pass it to `__Ownable_init`.

[L-02] DOS of user operations if totalXP is below 10

`_computeXP` calculates accrual rate modifier as `1e18/log10(totalXP)` if totalXP is greater than 0. The issue is that base10 log of any number less than 10 ranges from undefined, 0, to "0.decimals" which in solidity is eventually rounded down to 0. As a result, attempts to calculate `accrualRateModifier` will revert to dossing `_computeXP`.

```
function _computeXP(NoteXPData memory lastUpdate) internal view returns
      (uint256) {
        uint256 elapsed = block.timestamp - lastUpdate.lastAction;
        uint256 deposited = lastUpdate.keroseneDeposited;
        uint256 dyadMinted = lastUpdate.dyadMinted;
        uint256 totalXP = lastUpdate.totalXP;

        uint256 accrualRateModifier = totalXP > 0 ? 1e18 / totalXP.log10
        //() : 1e18; //@note

        uint256 adjustedAccrualRate = accrualRateModifier * 1e7;//@note
//...
```

As can be seen from the codebase, `_computeXP` is extensively in `updateXP` (called when dyad is minted/burned in the vaultmanager, tokens are deposited into the kerosene vaults, and when they are liquidated), `beforeKeroseneWithdrawn` (called when tokens are withdrawn from the kerosene vaults) and other instances of querying DyadXPv2 information.

The probability of this occurring is a bit low though as it requires a low amount of totalXP accumulated after kerosine withdrawal.

Recommend ensuring that totalXP is never below 10. The check can be refactored to > 100, since the log of values 10 - 99 is 1, and 1e18/1 is still 1e18

```
function _computeXP(NoteXPData memory lastUpdate) internal view returns
      (uint256) {
        uint256 elapsed = block.timestamp - lastUpdate.lastAction;
        uint256 deposited = lastUpdate.keroseneDeposited;
        uint256 dyadMinted = lastUpdate.dyadMinted;
        uint256 totalXP = lastUpdate.totalXP;

-       uint256 accrualRateModifier = totalXP > 0 ? 1e18 / totalXP.log10
- () : 1e18;
+       uint256 accrualRateModifier = totalXP > 99 ? 1e18 / totalXP.log10
+ () : 1e18;

        uint256 adjustedAccrualRate = accrualRateModifier * 1e7;
//...
```

[L-03] Unused `isDNftOwner` can be removed

Since the `isDNftOwner` modifier has been replaced with the `_authorizeCall` check. This modifier can be removed as it is unused.

[L-04] The unstake() function can be re-entered

Here is the order of operations:

```
_claimRewards(noteId, stakeInfo, recipient);

positionManager.safeTransferFrom(address(this), msg.sender, stakeInfo.tokenId);

delete stakes[noteId];
```

`safeTransferFrom()` will call the `onERC721Received()` function on `msg.sender` if `msg.sender` is a contract. At this point, since `stakes[noteId]` has not yet been deleted, the user can transfer the NFT back to the `UniswapV3Staking` contract, and then re-enter `unstake()` once again and it will not revert.

Since `_claimRewards()` updates `stakeInfo.lastRewardTime` to `block.timestamp`, the rewards earned upon reentrancy will be `0`, so the reentrancy does not cause any fund loss. However, it is risky because any future changes to the reward calculation can potentially lead to a critical vulnerability where an attacker re-enters unstake() to drain the rewards.

Currently the only impact is that the `Unstaked` event can be spammed an unlimited amount of times via this reentrancy.

The recomendation is to change the order of operations as follows:

```
_claimRewards(noteId, stakeInfo, recipient);

-positionManager.safeTransferFrom(address(this), msg.sender, stakeInfo.tokenId);

delete stakes[noteId];

+positionManager.safeTransferFrom(address(this), msg.sender, stakeInfo.tokenId);
```