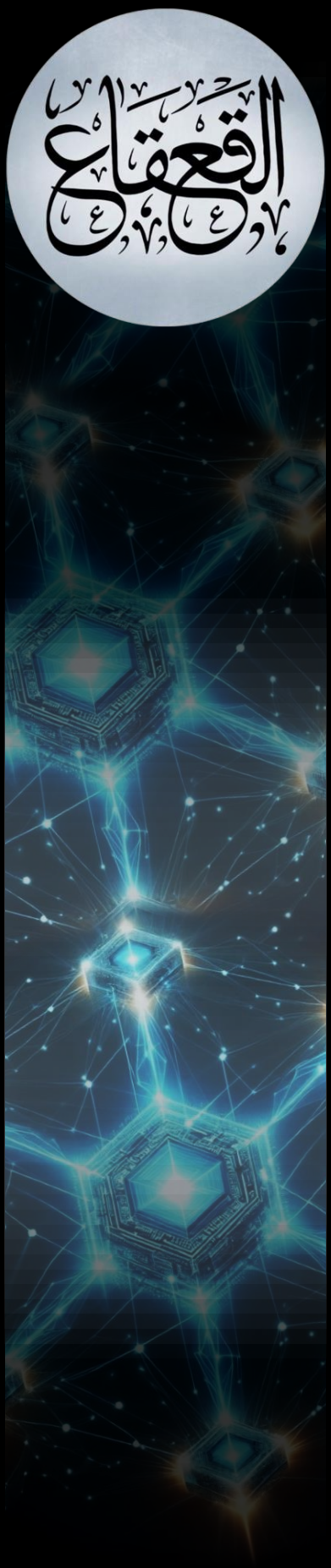
Al-Qa'qa’



DYAD

XPv2 Auditing Report

Auditor: Al-Qa'qa’

14 October 2024

Table of Contents

1 Introduction

1.1 About Al-Qa'qa'

Al-Qa'qa' is an independent Web3 security researcher who specializes in smart contract audits. Success in placing top 5 in multiple contests on code4rena and sherlock. In addition to smart contract audits, he has moderate experience in core EVM architecture, geth.

For security consulting, reach out to him on Twitter - @Al_Qa_qa

1.2 About DYAD

DYAD is a stablecoin protocol (Dollar pegged), it allows `ERC721` positions, where positions can be traded on third markets. In addition to this, they introduce the kerosine token, which its value depends on the volume of minted DYAD, and the locked collaterals, so it is as valuable as the degree of DYAD's over-collateralization.

1.3 Disclaimer

Security review cannot guarantee `100%` the safeness of the protocol, In the Auditing process, we try to get all possible issues, and we can not be sure if we missed something or not.

Al-Qa'qa' is not responsible for any misbehavior, bugs, or exploits affecting the audited code or any part of the deployment phase.

And change to the code after the mitigation process, puts the protocol at risk, and should be audited again.

1.4 Risk Classification

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

1.4.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol's core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol's functionality
- Low - Funds are **not** at risk

1.4.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align or little-to-no incentive

2 Executive Summary

2.1 Overview

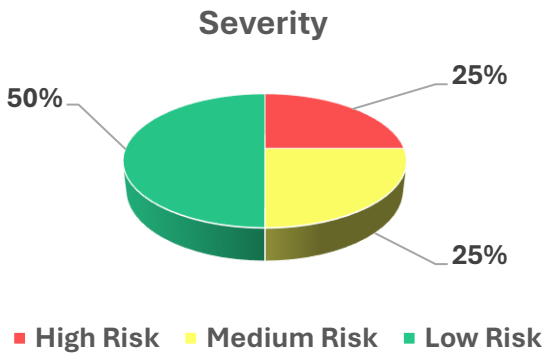
Project	**DYAD Stablecoin**
Repository	DyadStableCoin
Commit Hash	973cb961198890449e0a80b4be4065dccff0abc0
Mitigation Hash	5f31928caf31bbf8bcbb8e51316669394c7c9f86
Audit Timeline	14 Sept 2024 to 20 Sept 2024

2.2 Scope

- src/core/VaultManagerV5.sol
- src/staking/DyadXPv2.sol
- src/staking/UniswapV3Staking.sol

2.3 Issues Found

Severity	Count
High Risk	1
Medium Risk	1
Low Risk	2



3 Findings Summary

ID	Title	Severity
H-01	Staking Rewards calculations can be manipulated by narrowing Tick Boundaries	HIGH
M-01	`onlyOwner` modifier prevents Note owner to update his Balance himself	MEDIUM
L-01	`setHalvingConfiguration` checks are only done on first setting	LOW
L-02	Dividing Before Multiplications when calculating Reward, will give inaccurate results	LOW
I-01	Setting tick should insure `maxTick` is greater than `minTick`	INFO
I-02	using `storage` instead of `memory` for read only variables	INFO
I-03	LP tokens are forced to get transferred to the Note Owner	INFO
I-04	Removing a Vault Forces `CR` ratio checking for unLicensed Vaults	INFO

4 Findings

4.1 High Risk

4.1.1 Staking Rewards calculations can be manipulated by narrowing Tick Boundaries

context: UniswapV3Staking.sol#L144

Description
When calculating rewards for Stakers, there are a lot of variables that determines the value the staker will get from his position, and one of this parameters depends on His UniSwapV3 position, where we use liquidity variable to determine it.

UniswapV3Staking.sol#L144

```
    function _calculateRewards(uint256 noteId, StakeInfo storage stakeInfo)
internal view returns (uint256) {
        uint256 timeDiff = block.timestamp - stakeInfo.lastRewardTime;

        uint256 xp = dyadXP.balanceOfNote(noteId);

>>      (,,,,,,, uint128 liquidity,,,,) =
positionManager.positions(stakeInfo.tokenId);
>>      return timeDiff * rewardsRate * liquidity / 1e18 * xp / 1e18;
    }
```

The problem is that the liquidity value of a given position doesn't just depend on the Tokens added (token0/token1), but it also depends on the Tick Boundry Range.

UniV3::LiquidityAmounts.sol#L23-L31 | LiquidityAmounts.sol#L39-L46

```
    function getLiquidityForAmount0(
        uint160 sqrtRatioAX96,
        uint160 sqrtRatioBX96,
        uint256 amount0
    ) internal pure returns (uint128 liquidity) {
        if (sqrtRatioAX96 > sqrtRatioBX96) (sqrtRatioAX96, sqrtRatioBX96) =
(sqrtRatioBX96, sqrtRatioAX96);
        uint256 intermediate = FullMath.mulDiv(sqrtRatioAX96, sqrtRatioBX96,
FixedPoint96.Q96);
>>      return toUint128(FullMath.mulDiv(amount0, intermediate, sqrtRatioBX96 -
sqrtRatioAX96));
    }

    ...
    function getLiquidityForAmount1(
        uint160 sqrtRatioAX96,
        uint160 sqrtRatioBX96,
        uint256 amount1
    ) internal pure returns (uint128 liquidity) {
        if (sqrtRatioAX96 > sqrtRatioBX96) (sqrtRatioAX96, sqrtRatioBX96) =
(sqrtRatioBX96, sqrtRatioAX96);
>>      return toUint128(FullMath.mulDiv(amount1, FixedPoint96.Q96, sqrtRatioBX96
- sqrtRatioAX96));
    }
```

The liquidity is determined by the difference between the two `sqrtRatio` values, and these values depends in the Ticks. The more clother the min and max tick, the more the two values near each other.

LiquidityManagement.sol#L68-L69

```
    function addLiquidity(AddLiquidityParams memory params) ... {
        ...
        {
            (uint160 sqrtPriceX96, , , , , , ) = pool.slot0();
>>          uint160 sqrtRatioAX96 = TickMath.getSqrtRatioAtTick(params.tickLower);
>>          uint160 sqrtRatioBX96 = TickMath.getSqrtRatioAtTick(params.tickUpper);

            ...
        }
    }
```

So by providing too narrow tick bounries, the liquidity will increase significantly, and in case we expanded the boundary of Ticks, the liquidity will decrease.

This will make the value of `rate` unable to be set correctly, as if it is set for a current bondry like `100` difference between ticks, providing `50` difference between ticks will make the Staker earn twice his deserved price. And for the current Rate (17.36e9), it depends on a large boundary that even `100` difference will make Staking reward too huge even more than kerosine totalSuppy.

Proof of Concept
POC script in the first comment, by testing it we will find these results

After testing the issue, we get the following conclusion.

- current tick: `-276327`
- Trying adding 1000$ USD on each pair
- Forked Block Number 20_849_633 (not too old one)

==========

Mint [-276600, -276100] | (500 boundary)

- Liquidity: `0.07368e18`
- amount0: 830376514009102299068 (830$)
- amount1: 1000000000 (1000$)

Mint [-276400, -276300] | (100 boundary)

- Liquidity: `0.2732e18`
- amount0: 363944041864947375083 (364$)
- amount1: 1000000000 (1000$)

Mint [-276340, -276320] | (20 boundary)

- Liquidity: `1.5e18`
- amount0: 49893144176902108739624 (499$)
- amount1: 1000000000 (1000$)

As we can see the liquidity for `500` boundary between ticks is `0.07368e18` which costs `1830$`, and for `20` boundary the liquidity jumps up to `1.5e18` even with paying `1499$` dollar. If we assume the cost is the same, we can conclude that the

increase of liquidity is: 1.5 / 0.07368 ~= 20. Which means the staking reward will increased 20 times for that Staker. and we should make sure that the increase here is exponential, narrowing more and more will make liquidity increase significantly.

Recommendations
We need to make replace liquidity with the value of tokens in a given position. For example, if the position have 1000e6 USD, and 500e18 DYAD, we can make the parameter is `1500e18`. This will make the positions with High Amounts of tokens getes more rewards.

Since UniSwapStaking doesn't accumulate position rewards, and the staker can claim it only if he unstake it, it will affect the user himself, to determine the ticks in a profitable range to him.

To get the amount of token using liquidity we can use `getAmount0ForLiquidity()` and `getAmount1ForLiquidity()` logic in UniswapV3, which takes liquidity and gives you the amount of tokens.

4.2 Medium Risk

4.2.1 `onlyOwner` modifier prevents Note owner to update his Balance himself

context: DyadXPv2.sol#L138-L145

Description
In the new version of DyadXP, there is a function that can be called to auto modify Note XP without performing depositing/withdrawing etc... from the VaultManager. This function is designed to be only called by Protoocl Admins or the owner of the Note, to update his balance. The problem is that Although the function is designed to be callable by Note owner, if Note Owner calls it it will revert because of the presence of `onlyOwner` modifier which will make the tx revert.

DyadXPv2.sol#L138-L145

```
    function forceUpdateXPBalance(uint256 noteId) external onlyOwner {
        if (msg.sender != owner()) {
            if (msg.sender != DNFT.ownerOf(noteId)) {
                revert Unauthorized();
            }
        }
        _updateNoteBalance(noteId);
    }
```

The function checks that the caller is the protocol owner; if not, it checks that the caller is the Note owner. But as `onlyOwner` modifier exists, calling it by Note owner will revert.

This will prevent Note Holders from updating their balance manually, which is a supported feature in the version of DyadXP.

Recommendations
Remove `onlyOwner` modifier from the function.

4.3 Low Risk

4.3.1 `setHalvingConfiguration` checks are only done on first setting

context: DyadXPv2.sol#L191-L200

Description
When setting new Halving we use `setHalvingConfiguration` function, there are some checks we should make to ensure the configurations are setted correctly.

```
    function setHalvingConfiguration( ... ) external onlyOwner {
        if (halvingStart != 0) {
            uint256 dnftSupply = DNFT.totalSupply();
            for (uint256 i = 0; i < dnftSupply; ++i) {
                _updateNoteBalance(i);
            }
        } else if (_halvingStart < halvingStart) {
            revert InvalidConfiguration();
        } else if (_halvingStart < block.timestamp) {
            revert InvalidConfiguration();
        }
        ...
    }
```

We have `if`, `else if`, `else if`, so in case one condition is met the other will not checks.

Since this is the second version of DyadXP, and Havling logic is new in V2, halving configurations will be `0`.

This will make the first setting escape the first `if`, and go for `else if`.

It will check that the new Havling is not before the old Having and it comes after the current `block.timestamp`. But since these checks are in `else if`, they will only be checked if we escaped the first `if` condition.

So in case the first setting of halving occurs, and after some time, we make another setting, the `halvingStart` will not be `0` so will go for Updating NoteXPs from the first if condition. But since we goes for the first `if` condition. the security checks for `halvingStart` will not occur.

Recommendations
Check that `halvingStart` is greater than the previous havingStart in addition to being greaterthan or equal `block.timestamp`

```
diff --git a/src/staking/DyadXPv2.sol b/src/staking/DyadXPv2.sol
index 44b5d69..71d06cb 100644
--- a/src/staking/DyadXPv2.sol
+++ b/src/staking/DyadXPv2.sol
@@ -188,18 +188,18 @@ contract DyadXPv2 is IERC20, UUPSUpgradeable,
OwnableUpgradeable {
         uint40 _halvingStart,
         uint40 _halvingCadence
     ) external onlyOwner {
+        if (
```

```
+            (_halvingStart != 0 && (_halvingStart < halvingStart || _halvingStart
< block.timestamp)) ||
+            (_halvingStart != 0 && _halvingCadence == 0)
+        ) {
+            revert InvalidConfiguration();
+        }
+
         if (halvingStart != 0) {
             uint256 dnftSupply = DNFT.totalSupply();
             for (uint256 i = 0; i < dnftSupply; ++i) {
                 _updateNoteBalance(i);
             }
-        } else if (_halvingStart < halvingStart) {
-            revert InvalidConfiguration();
-        } else if (_halvingStart < block.timestamp) {
-            revert InvalidConfiguration();
-        }
-        if (_halvingCadence == 0) {
-            revert InvalidConfiguration();
         }

         halvingStart = _halvingStart;
```

This modification will do the following.

- Allows preventing halving configurations, by setting halving start time to `0`, and `halvingCadence` to `0` too.
- In case setting a new halving config, we check that the new starting time, is greater than the previous starting time, and greater than or equal `block.timestamp` as it should not start in the past
- If there was a halving setting before, updating all Note XP to give All Notes the XP they deserve in the period of halving
- Preventing `halvingCadence` from being `zero`, in case havling is supported

4.3.2 Dividing Before Multiplications when calculating Reward, will give inaccurate results

contest: UniswapV3Staking.sol#L144

Description
When calculating Rewards for a given Staked Note, we multiply the time elapsed with the liquidity and Note XP, by doing a rate and some divisions.

UniswapV3Staking.sol#L144

```
    function _calculateRewards(uint256 noteId, StakeInfo storage stakeInfo)
internal view returns (uint256) {
        ...
>>      return timeDiff * rewardsRate * liquidity / 1e18 * xp / 1e18;
    }
```

We are multiplying `timeDiff` by `rate` then `liquidity`, then dividing by `1e18` and then do another multiplication by `xp`, which will result in truncating huge decimals, from the liquidity, as it is not multiplied by XP. And the problem is that in case the liquidity is too small, with a small Rate. it can result in Getting zero value.

It is always best practice to do multiplication before dividing to get the correct result and avoid truncating of most of the number by division.

Recommendations
Make the Division the last step after doing all the multiplications.

```
diff --git a/src/staking/UniswapV3Staking.sol b/src/staking/UniswapV3Staking.sol
index 657d135..91909e7 100644
--- a/src/staking/UniswapV3Staking.sol
+++ b/src/staking/UniswapV3Staking.sol
@@ -141,7 +141,7 @@ contract UniswapV3Staking is UUPSUpgradeable,
OwnableUpgradeable {
         uint256 xp = dyadXP.balanceOfNote(noteId);

         (,,,,,,, uint128 liquidity,,,,) =
positionManager.positions(stakeInfo.tokenId);
-        return timeDiff * rewardsRate * liquidity / 1e18 * xp / 1e18;
+        return timeDiff * rewardsRate * liquidity * xp / 1e36;
     }

     function currentRewards(uint256 noteId) external view returns (uint256) {
```

4.4 Informational Findings

4.4.1 Setting tick should insure `maxTick` is greater than `minTick`

context: UniswapV3Staking.sol#L167

Description
When setting `minTick` and `maxTick`, the check implemented insure that `maxTick` is greater than or equal `minTick`.

UniswapV3Staking.sol#L167

```
    function setTickRange(int24 _minTick, int24 _maxTick) external onlyOwner {
>>      require(_minTick <= _maxTick, "Invalid tick range");
        minTick = _minTick;
        maxTick = _maxTick;
    }
```

This is not the correct check implemented as in UniswapV3 the maxTick should be greater than minTick, and can't equal it.

UniV3::UniswapV3Pool.sol#L127

```
    function checkTicks(int24 tickLower, int24 tickUpper) private pure {
>>      require(tickLower < tickUpper, 'TLU');
        require(tickLower >= TickMath.MIN_TICK, 'TLM');
        require(tickUpper <= TickMath.MAX_TICK, 'TUM');
    }
```

Recommendations
Make the check greater than instead of greater than or equal, to match UniswapV3 check.

```
diff --git a/src/staking/UniswapV3Staking.sol b/src/staking/UniswapV3Staking.sol
index 657d135..93a4d6f 100644
--- a/src/staking/UniswapV3Staking.sol
+++ b/src/staking/UniswapV3Staking.sol
@@ -164,7 +164,7 @@ contract UniswapV3Staking is UUPSUpgradeable,
OwnableUpgradeable {
     }

     function setTickRange(int24 _minTick, int24 _maxTick) external onlyOwner {
-        require(_minTick <= _maxTick, "Invalid tick range");
+        require(_minTick < _maxTick, "Invalid tick range");
         minTick = _minTick;
         maxTick = _maxTick;
     }
```

4.4.2 using `storage` instead of `memory` for read-only variables

context: UniswapV3Staking.sol#L72

Description
When doing stake, we are checking that the current NoteId is not already staked. we store the stake using `storage` word.

UniswapV3Staking.sol#L72

```
    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");

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

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

When modifying the stake mapping we are modifying the mapping directly, without using `stakeInfo` storage variable. So making the variable `storage` is useless as we only use it for reading, and not writing.

Recommendations
Replace `storge` word with `memory`

4.4.3 LP tokens are forced to get transferred to the Note Owner

context: UniswapV3Staking.sol#L115

Description
Each note Holder can participate in Staking by only staking one Liquidity position token for UniV3. when doing the staking, we are transferring the token from the NoteOwner to `UniswapV3Staking` address.

UniswapV3Staking.sol#L98

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

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

        ...
    }
```

In case of NFTs, the `msg.sender` can be the owner or not. as it can be one of the approvals addresses that have the ability to transfer that token, or even all owners tokens. So When NoteOwner transfers `LP` NFT this doesn't mean he is the owner of it.

Now in case of unstaking, the NFT is transferred back to the `msg.sender`, and as we illustrated it can be the Note owner decision to not hold that position, as it may even be not the owner when he first staking.

[UniswapV3Staking.sol#L115]

```
    function unstake(uint256 noteId, address recipient) external {
        ...
        delete stakes[noteId];

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

This will prevent Note Owners how don't own there LP tokens and just have approvals to them, being unable to unstake tokens, and transfer them back to the original Owner, because it is transferred to `msg.sender` which is the Note owner.

Recommendations

Allow providing `positionReceipent` address, that will receive the LP token, instead of forcing to `msg.sender`

```
diff --git a/src/staking/UniswapV3Staking.sol b/src/staking/UniswapV3Staking.sol
index 657d135..0774ae4 100644
--- a/src/staking/UniswapV3Staking.sol
+++ b/src/staking/UniswapV3Staking.sol
@@ -103,16 +103,16 @@ contract UniswapV3Staking is UUPSUpgradeable,
OwnableUpgradeable {
         emit Staked(msg.sender, noteId, tokenId, liquidity);
     }

-    function unstake(uint256 noteId, address recipient) external {
+    function unstake(uint256 noteId, address rewardRecipient, address
positionRecipient) external {
         StakeInfo storage stakeInfo = stakes[noteId];

-        _claimRewards(noteId, stakeInfo, recipient);
+        _claimRewards(noteId, stakeInfo, rewardRecipient);

         uint tokenId = stakeInfo.tokenId;

         delete stakes[noteId];

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

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

4.4.4 Removing a Vault Forces `CR` ratio checking for unLicensed Vaults

context: VaultManagerV5.sol#L88

Description
Dyad Supports Vaults that can be used to add Collateral to mint Dyad tokens. These Vaults should be licenses for the collateral to be used in calculating the total USD value of a given Note.

VaultManagerV5.sol#L307-L313

```
  function getVaultsValues( ... ) ... {
      ...
>>      if (vaultLicenser.isLicensed(address(vault))) {
          if (vaultLicenser.isKerosene(address(vault))) {
            keroValue += vault.getUsdValue(id);
          } else {
            exoValue  += vault.getUsdValue(id);
          }
        }
      }
  }
```

So the Vaults that are not Licenced don't even gets in the calculation of the `CR` of a given Note.

When removing a Vault, and this Vault has assets, we check for the `CR` of that Note, but there is no check whether this Vault is actually a licensed Vault.

VaultManagerV5.sol#L88

```
  function remove( ... ) ... {
    _authorizeCall(id);
    if (vaults[id].remove(vault)) {
>>    if (Vault(vault).id2asset(id) > 0) {
        _checkExoValueAndCollatRatio(id);
      }
      emit Removed(id, vault);
    }
  }
```

So In case of unLicencing a given vault from the Owner, the Users will not be able to remove their vault in case the `CR` is undercollateralized, even in case the Vault is not supported.

Recommendations
Check fro the `CR` ration in case the Vault is Licenced.

```
diff --git a/src/core/VaultManagerV5.sol b/src/core/VaultManagerV5.sol
index b545117..bc8bc0d 100644
--- a/src/core/VaultManagerV5.sol
+++ b/src/core/VaultManagerV5.sol
@@ -85,7 +85,7 @@ contract VaultManagerV5 is IVaultManagerV5, UUPSUpgradeable,
OwnableUpgradeable
   {
     _authorizeCall(id);
```

```
     if (vaults[id].remove(vault)) {
-      if (Vault(vault).id2asset(id) > 0) {
+      if (Vault(vault).id2asset(id) > 0 && vaultLicenser.isLicensed(vault)) {
         _checkExoValueAndCollatRatio(id);
       }
       emit Removed(id, vault);
```