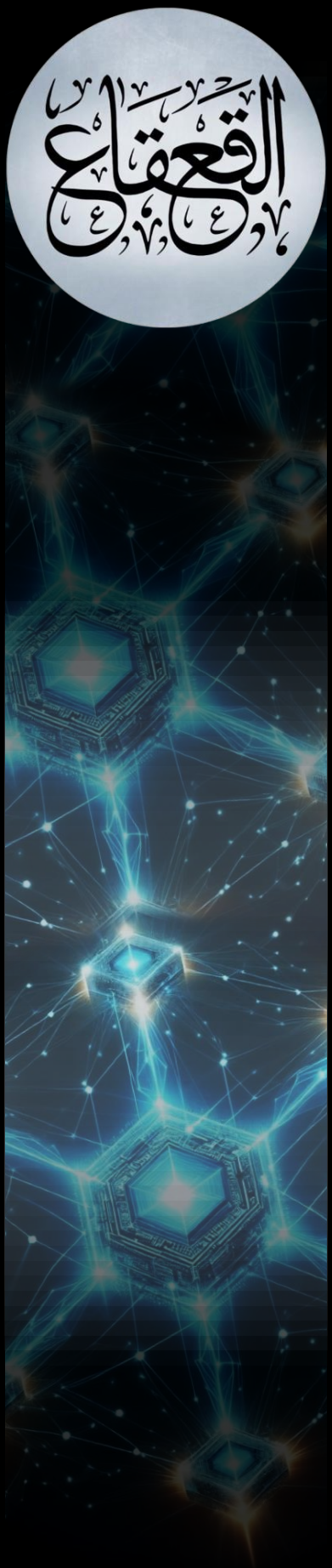
Al-Qa'qa’



DYAD

DyadLpStaking Auditing Report

Auditor: Al-Qa'qa’

27 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

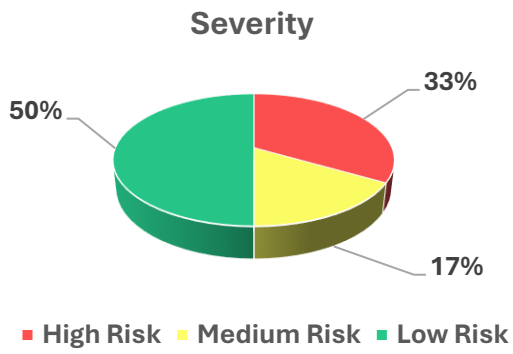
Project	**DYAD Stablecoin**
Repository	DyadStableCoin::feat/erc20-lp-staking
Commit Hash	b76cf79afdb2c68bc4f432597c593ab9a29a65b4
Mitigation Hash	8b7b1e34171c348d057c4fc3cdd2c96513776911
Audit Timeline	17 Oct 2024 to 23 Oct 2024

2.2 Scope

- src/staking/DyadLPStakingFactory.sol
- src/staking/DyadLPStaking.sol
- src/periphery/AtomicSwapExtension.sol
- src/periphery/RedeemCollateralExtension.sol

2.3 Issues Found

Severity	Count
High Risk	2
Medium Risk	1
Low Risk	3



3 Findings Summary

ID	Title	State
H-01	`swapCollateral` Lacks Note Owners Access Control	Fixed
H-02	Memory Pointer is Updated incorrectly	Fixed
M-01	Storing `swapData` into memory is implemented incorrectly	Fixed
L-01	There is no event emiting on changing Critical variables	Fixed
L-02	`depositForRewards()` is not checking for pausing of the contracts	Fixed
L-03	`DyadLPStaking::deposit()` is not checking the validity of dNFT	Fixed
I-01	pragma is missing in `DyadLPStakingFactory` contract	Fixed
I-02	Unused contracts logic is inherited by `DyadLpStaking`	Fixed
I-03	Recovering ERC20 tokens should exclude Kerosine in `DyadLPStakingFactory`	Fixed
I-04	Claiming tokens is not checking zero amount claim	Fixed
I-05	Remove declared variable that are not used	Fixed
I-06	remove useless imports from the Codebase	Fixed

4 Findings

4.1 High Risk

4.1.1 `swapCollateral` Lacks Note Owners Access Control

context: AtomicSwapExtension.sol#L37-L44

Description
`AtomicSwapExtension` enables users to change their Collaterals, Users enabling this extension should be used to do that process.

In `AtomicSwapExtension::swapCollateral()` the function withdraw collaterals from Vault, and then the process completed via `afterWithdraw()` function. But the problem is that `swapCollateral()` is not checking that the caller is the Note Owner himself

AtomicSwapExtension.sol#L37-L44

```
    function swapCollateral( ... ) external {
        // @audit `noteOwner == msg.sender` is missing
        // tstore swap data
        assembly {
            let size := calldatasize()
            let slot := 0
            for { let i := 0x64 } lt(i, size) { i := add(i, 0x20) } {
                tstore(slot, calldataload(i))
                slot := add(slot, 1)
            }
        }

        vaultManager.withdraw(noteId, fromVault, fromAmount, address(this));
    }
```

So anyone can call that function at any `noteId` that its user allowed that extensions.

This will make anyone have the ability to change the collateral of other Notes he don't have access too. and the process can be more critical, as in High Slippage markets, the attacker can make the position lose by making a losing Swap, etc...

Recommendations
Check that the caller is the `note` owner.

_NOTE: you need to add `dnft` address itself in the constructor

Sponsor: Fixed at PR-120,
commit: 8997d434dea5985e058c649a953aae51ce800718
Al-Qa'qa': Verified. The issue has been fixed as recommended.

4.1.2 Memory Pointer is Updated incorrectly

context:

- AtomicSwapExtension.sol#L85
- Mitigating `H-01` will make the function succeed with data truncation results in completely incorrect results.

Description
After fetching Data from Transit Storage and putting it in the memory using `mstore`, we are updating the memory pointer, but updating it is done by a completely incorrect method.

AtomicSwapExtension.sol#L85

```
        assembly {
            ...
1:          swapData := mload(0x40)
            let i := 1
            let swapDataSize := tload(3)
2:          mstore(swapData, swapDataSize) // memory changed
            for {} lt(sub(i, 1), swapDataSize) { i := add(i, 1) } {
3:              mstore(add(swapData, mul(i, 0x20)), tload(add(i, 3))) // memory
changed
            }
4:          mstore(0x40, add(swapData, mul(sub(i, 1), 0x20))) // update memory
pointer
        }
```

After we Store the memory pointer in `swapData`, we load the `swapData` from transit storage, and after finish setting the memory pointer to the new value.

The problem here is that we are using Slot based addition to update our memory pointer.

If we are going to store `0x24` Bytes in memory, the memory will have the following values.

- [0x40]: Free memory Pointer (FMP)
- [FMP Slot]: Bytes length (0x24)
- [FMP + 1 Slot]: first `0x20` bytes
- [FMP + 2 Slot]: first [0x21, 0x24] data value (store only 4 bytes)
- Know if we should update memory pointer
- new memory pointer: old memory pointer + 0x20 (for the length value) + 0x24 (data itself)
- [0x40]: new memory pointer value

This is the correct way to update the free memory pointer, but in our implementation, we are doing two issues

1. We are not including the length of bytes, we just add the values of the bytes to the old memory pointer value, without taking into consideration the length of bytes that will take a complete slot `0x20`

2. We are updating the `FMP` using iteration value, which is not correct the value in the last transit slot can be incomplete. Where if The data is a function with just one `uint256` parameter, the length of the byte is `0x24`. So the 2nd transit slot will only contain the last 4 bytes of data, but we are updating the value of `FMP` by increasing the total `0x20`.

This will make the last data values get truncated (padded by zeros) as the `FMP` will prevent reading the values.

Recommendations

Update the `FMP` using the DataSize instead of depending on Slots, and add `0x20` for the Data length Slot.

```
diff --git a/src/periphery/AtomicSwapExtension.sol
b/src/periphery/AtomicSwapExtension.sol
index 6e3ade1..5e8365a 100644
--- a/src/periphery/AtomicSwapExtension.sol
+++ b/src/periphery/AtomicSwapExtension.sol
@@ -82,7 +82,7 @@ contract AtomicSwapExtension is IExtension, IAfterWithdrawHook {
                 mstore(add(swapData, mul(i, 0x20)), tload(add(i, 3)))
             }
             // update the free memory pointer
-            mstore(0x40, add(swapData, mul(sub(i, 1), 0x20)))
+            mstore(0x40, add(swapData, add(swapDataSize, 0x20)))
         }

         IVault(vault).asset().safeApprove(AUGUSTUS_6_2, amount);
```

Sponsor: Fixed at PR-122,
commit: 476f86c5b0fb0b642202791384d05696bd2e805c

Al-Qa'qa': Verified. The issue has been fixed as recommended.

4.2 Medium Risk

4.2.1 Storing swapData into memory is implemented incorrectly

context: AtomicSwapExtension.sol#L80-L83

Description

When doing our AtomicSwap, we store `swapData` in transit storage, then we read it and store it in the memory to do our call.

AtomicSwapExtension.sol#L80-L83

```
        assembly {
            ...
            let swapDataSize := tload(3)
            // store the size of the swapData in the first slot of the memory
array
            mstore(swapData, swapDataSize)
            // iterate over the addresses and copy them into the memory array
>>          for {} lt(sub(i, 1), swapDataSize) { i := add(i, 1) } {
                // copy from transient storage into memory
                mstore(add(swapData, mul(i, 0x20)), tload(add(i, 3)))
            }
```

```
            // update the free memory pointer
            mstore(0x40, add(swapData, mul(sub(i, 1), 0x20)))
        }
```

We initialize the iterator `i` by one, and each loop we increase its size by `1` till it reach the `swapDataSize` value.

The problem is that there is a difference between storing storage numbers and `swapDataSize`.

- `swapDataSize` represents the length of bytes, i.e each byte increases the size by `1`. So in case we have a function signature with a single `uint256` variable, we will have `swapDataSize` equals 32 + 4 = 36 byte.
- In the contrary, when dealing with `load/store`, we are dealing with slots, not bytes, i.e each slot represent `32` bytes.

Our iteration relies on bytes length, which will make it repeat `36` times in case of our example, but the `store/load` will end in the 2nd process.

The loop will still process loading zero bytes32 value from transit storage and store them in memory. with no need,making the Gas Cost increases significantly, which may lead to OOG in case of large sized `swapData`.

Since we are dealing with `ParaSwap`, the functions take a lot of parameters, which will make the `swapData` size large, making the for loop contain to iterate, and can lead to OOG issues.

Recommendations
Instead of checking `i` against `swapDataSize` check `i multiply 0x20`

```
diff --git a/src/periphery/AtomicSwapExtension.sol
b/src/periphery/AtomicSwapExtension.sol
index 6e3ade1..80ffc42 100644
--- a/src/periphery/AtomicSwapExtension.sol
+++ b/src/periphery/AtomicSwapExtension.sol
@@ -77,7 +77,7 @@ contract AtomicSwapExtension is IExtension, IAfterWithdrawHook {
             // store the size of the swapData in the first slot of the memory
array
             mstore(swapData, swapDataSize)
             // iterate over the addresses and copy them into the memory array
-            for {} lt(sub(i, 1), swapDataSize) { i := add(i, 1) } {
+            for {} lt(mul(sub(i, 1), 0x20), swapDataSize) { i := add(i, 1) } {
                 // copy from transient storage into memory
                 mstore(add(swapData, mul(i, 0x20)), tload(add(i, 3)))
             }
```

Sponsor: Fixed at PR-121,
commit: 55ab7f8a12188ed6c63bd8a99b4e20a05e2ac242

Al-Qa'qa': Verified. The issue has been fixed as recommended.

4.3 Low Risk

4.3.1 There is no event emitting on changing Critical variables

context:

- DyadLPStakingFactory.sol#L102-L104
- DyadLPStakingFactory.sol#L140-L143

Description
The Staking Mechanism relies heavily on off-chain system, so emitting events is crucial for monitoring changes occur to `DyadLPStakingFactory` contract.

When changing pausing variable in `DyadLPStakingFactory` there is no event emitting.

DyadLPStakingFactory.sol#L102-L104

```
    function setPaused(bool _paused) public onlyOwnerOrRoles(POOL_MANAGER_ROLE) {
        paused = _paused;
    }
```

And when setting the percentage of fees that will be taken when withdrawing kerosine tokens directly, by changing `directDepositBonusBps`, there is no event emits too.

DyadLPStakingFactory.sol#L140-L143

```
    function setDirectDepositBonus(uint16 _directDepositBonusBps) public
onlyOwnerOrRoles(REWARDS_MANAGER_ROLE) {
        require(_directDepositBonusBps <= 10000, InvalidBonus());
        directDepositBonusBps = _directDepositBonusBps;
    }
```

This will make analyzing the changes occur to the Staking Contracts harder, which is used to distribute rewards to users, etc...

Recommendations
Emit events when firing `setPaused()` and `setDirectDepositBonus()` functions

Sponsor: Fixed at PR-118,
commit: 709e0e5761ecf0a7446227a099b5b36a6d4c3047, edbe2b124d07aad82300e91ade8f8781ce8bb9de, 2d99879b70144b557ca8817e459a17eeaa8b40f8

Al-Qa'qa': Verified. The issue has been fixed as recommended.

4.3.2 `depositForRewards()` is not checking for pausing of the contracts

context: DyadLPStakingFactory.sol#L120

Description
`DyadLPStakingFactory` have pausing mechanism, that will result in stopping of rewards claiming by Stakers, by preventing `claim()`and `claimToVault()` functions.

There is another function that is used to add rewards to the system itself, which is `depositForRewards()`, this function is not checking for the pausing status, which means it can distribute rewards even if the staking is paused.

DyadLPStakingFactory.sol#L120

```
    function depositForRewards(uint256 amount) public
onlyOwnerOrRoles(REWARDS_MANAGER_ROLE) {
        ...
    }
```

Recommendations
Check for pausing of staking before distributing rewards.

```
diff --git a/src/staking/DyadLPStakingFactory.sol
b/src/staking/DyadLPStakingFactory.sol
index 439bf27..3d2aa48 100644
--- a/src/staking/DyadLPStakingFactory.sol
+++ b/src/staking/DyadLPStakingFactory.sol
@@ -117,7 +117,7 @@ contract DyadLPStakingFactory is OwnableRoles, IExtension {
         }
     }

-    function depositForRewards(uint256 amount) public
onlyOwnerOrRoles(REWARDS_MANAGER_ROLE) {
+    function depositForRewards(uint256 amount) public
onlyOwnerOrRoles(REWARDS_MANAGER_ROLE) whenNotPaused {
         uint256 previousUnclaimedBonus = unclaimedBonus;
         unclaimedBonus = 0;
         if (amount < previousUnclaimedBonus) {
```

Sponsor: Fixed at PR-117,
commit: 854988d70aa7e829bb2a0e0a23c403c9a6fa6ca6

Al-Qa'qa': Verified. The issue has been fixed as recommended.

4.3.3 `DyadLPStaking::deposit()` is not checking the validity of dNFT

context: DyadLPStaking.sol#L37-L43

Description
when users deposit tokens into a given LP token staking contract, and provide the `noteId`, there is not check weather this `noteId` is valid or not.

DyadLPStaking.sol#L37-L43

```
    function deposit(uint256 noteId, uint256 amount) public {
        totalLP += amount;
        noteIdToAmountDeposited[noteId] += amount;
        lpToken.safeTransferFrom(msg.sender, address(this), amount);

        emit Deposited(noteId, amount);
    }
```

This will allow users to stake for positions that haven't yet been created, besides the lack of robustness of the code, and compatibility with VaultManagerV5 that prevents depositing into an Invalid `NoteId`.

Recommendations
Check that the `NoteId` is valid, before depositing.

```
diff --git a/src/staking/DyadLPStaking.sol b/src/staking/DyadLPStaking.sol
index 534d402..30b2af2 100644
--- a/src/staking/DyadLPStaking.sol
+++ b/src/staking/DyadLPStaking.sol
@@ -35,6 +35,7 @@ contract DyadLPStaking is OwnableRoles {
     }

     function deposit(uint256 noteId, uint256 amount) public {
+        require(dnft.ownerOf(noteId) != address(0), "Invalid NoteId");
         totalLP += amount;
         noteIdToAmountDeposited[noteId] += amount;
         lpToken.safeTransferFrom(msg.sender, address(this), amount);
```

Sponsor: Fixed at PR-117,
commit: 854988d70aa7e829bb2a0e0a23c403c9a6fa6ca6

Al-Qa'qa': Verified. The issue has been fixed as recommended.

4.4 Informational Findings

4.4.1 pragma is missing in `DyadLPStakingFactory` contract

context: DyadLPStakingFactory.sol#L1

Description
Each solidity file `.sol` should have a pragma interface that determines the version uses with the Licence, but it is not in `staking/DyadLPStakingFactory.sol` file. It starts directly with imports.

```
import {Ownable} from "solady/auth/Ownable.sol";
import {DyadLPStaking} from "./DyadLPStaking.sol";
```

Recommendations
Put the `pragma` definition in the beginning of the file.

```
diff --git a/src/staking/DyadLPStakingFactory.sol
b/src/staking/DyadLPStakingFactory.sol
index 439bf27..45e9739 100644
--- a/src/staking/DyadLPStakingFactory.sol
+++ b/src/staking/DyadLPStakingFactory.sol
@@ -1,3 +1,6 @@
+// SPDX-License-Identifier: MIT
+pragma solidity ^0.8.27;
+
 import {Ownable} from "solady/auth/Ownable.sol";
 import {DyadLPStaking} from "./DyadLPStaking.sol";
 import {EnumerableSetLib} from "solady/utils/EnumerableSetLib.sol";
```

Sponsor: Fixed at PR-118,
commit: 709e0e5761ecf0a7446227a099b5b36a6d4c3047

Al-Qa'qa': Verified. The issue has been fixed as recommended.

4.4.2 Unused contracts logic is inherited by `DyadLpStaking`

context: DyadLpStaking.sol#L11

Description

`DyadLpStaking` inherits from `Soloday::OwnableRoles` contract, which is used to represent Roles besides the Ownership of the contracts.

`OwnableRoles` inherits from `Ownable`, and there is no use for Rolling mechanism in `DyadLpStaking.sol`, making contract size larger with no need.

Recommendations

use `Ownable` instead of `OwnableRoles` in `DyadLpStaking.sol`

```
diff --git a/src/staking/DyadLPStaking.sol b/src/staking/DyadLPStaking.sol
index 534d402..dd9e4fb 100644
--- a/src/staking/DyadLPStaking.sol
+++ b/src/staking/DyadLPStaking.sol
@@ -4,11 +4,11 @@ pragma solidity ^0.8.27;
 import {IERC20} from "forge-std/interfaces/IERC20.sol";
 import {IERC721} from "forge-std/interfaces/IERC721.sol";
 import {SafeTransferLib} from "solady/utils/SafeTransferLib.sol";
-import {OwnableRoles} from "solady/auth/OwnableRoles.sol";
+import {Ownable} from "solady/auth/Ownable.sol";
 import {MerkleProofLib} from "solady/utils/MerkleProofLib.sol";
 import {FixedPointMathLib} from "solady/utils/FixedPointMathLib.sol";

-contract DyadLPStaking is OwnableRoles {
+contract DyadLPStaking is Ownable {
     using SafeTransferLib for address;
     using FixedPointMathLib for uint256;
```

Sponsor: Fixed at PR-118,
commit: 709e0e5761ecf0a7446227a099b5b36a6d4c3047

Al-Qa'qa': Verified. The issue has been fixed as recommended.

4.4.3 Recovering ERC20 tokens should exclude Kerosine in `DyadLPStakingFactory`

context: DyadLPStakingFactory.sol#L201-L204

Description

In `DyadLPStakingFactory` contract, there are recovery functions that allow the Owner to take any ERC20 token or ERC721 token transferred wrongly to the contract. Since the contract uses kerosine tokens in depositing, it will contain a supply of kerosine tokens.

DyadLPStakingFactory.sol#L201-L204

```
    function recoverERC20(address token) public onlyOwner {
        uint256 amount = IERC20(token).balanceOf(address(this));
        token.safeTransfer(msg.sender, amount);
    }
```

Since these Kerosine tokens are claimable by stakers, there is no need to recover them back by the Owner who distribute them in the first place, and even in case of incorrect sending kerosine tokens by mistake, it is not a big deal it will make the Owner distribute less reward instead.

Recommendations
Prevent recovering kerosine tokens.

```
diff --git a/src/staking/DyadLPStakingFactory.sol
b/src/staking/DyadLPStakingFactory.sol
index 439bf27..f714207 100644
--- a/src/staking/DyadLPStakingFactory.sol
+++ b/src/staking/DyadLPStakingFactory.sol
@@ -199,6 +199,7 @@ contract DyadLPStakingFactory is OwnableRoles, IExtension {
     }

     function recoverERC20(address token) public onlyOwner {
+        require(token != kerosene, "Can't Recover Kerosine Token");
         uint256 amount = IERC20(token).balanceOf(address(this));
         token.safeTransfer(msg.sender, amount);
     }
```

Sponsor: Fixed at PR-118,
commit: 709e0e5761ecf0a7446227a099b5b36a6d4c3047

Al-Qa'qa': Verified. The issue has been fixed as recommended.

4.4.4 Claiming tokens is not checking zero amount claim

context:

- DyadLPStakingFactory.sol#L152-L158
- DyadLPStakingFactory.sol#L173-L177

Description
When claiming Rewards, there is no check if the reward is zero or not, the Kerosine Token transfer is fired even if there is no reward that Note owner has.

DyadLPStakingFactory.sol#L152-L158 | DyadLPStakingFactory.sol#L173-L177

```
    function claim(uint256 noteId, uint256 amount, bytes32[] calldata proof)
public whenNotPaused returns (uint256) {
        ...
>>      uint256 amountToSend = _syncClaimableAmount(noteId, amount);
        uint256 claimSubBonus = amountToSend.mulDiv(10000 - bonusBps, 10000);
        uint256 unclaimed = amountToSend - claimSubBonus;
        unclaimedBonus += uint96(unclaimed);
        totalClaimed += uint128(claimSubBonus);

>>      kerosene.safeTransfer(noteOwner, claimSubBonus);
       ...
```

```
    }

    ...

    function claimToVault( ... ) ... {
        ...
>>      uint256 amountToSend = _syncClaimableAmount(noteId, amount);
        totalClaimed += uint128(amountToSend);

        kerosene.safeApprove(address(vaultManager), amountToSend);
>>      vaultManager.deposit(noteId, keroseneVault, amountToSend);
        ...
    }
```

This will make `claimToVault` go for approving, then depositing to `VaultManager`, which will call `DyadXP`, updating Note balance, using zero amount of tokens, which is not a good scenario for emitting such events for interactions that doesn't changed the State in reality.

Since the reward is taken by the Owner as one time using `MerkleProof`, the staker can forget and reclaim again. which increases the possibility of the occurrence of such scenario.

Recommendations
Check that the reward is greater than `0`

```
diff --git a/src/staking/DyadLPStakingFactory.sol
b/src/staking/DyadLPStakingFactory.sol
index 439bf27..149ede3 100644
--- a/src/staking/DyadLPStakingFactory.sol
+++ b/src/staking/DyadLPStakingFactory.sol
@@ -150,6 +150,7 @@ contract DyadLPStakingFactory is OwnableRoles, IExtension {

         _verifyProof(noteId, amount, proof);
         uint256 amountToSend = _syncClaimableAmount(noteId, amount);
+        require(amountToSend > 0, "Reward Already Claimed");
         uint256 claimSubBonus = amountToSend.mulDiv(10000 - bonusBps, 10000);
         uint256 unclaimed = amountToSend - claimSubBonus;
         unclaimedBonus += uint96(unclaimed);
@@ -171,6 +172,7 @@ contract DyadLPStakingFactory is OwnableRoles, IExtension {

         _verifyProof(noteId, amount, proof);
         uint256 amountToSend = _syncClaimableAmount(noteId, amount);
+        require(amountToSend > 0, "Reward Already Claimed");
         totalClaimed += uint128(amountToSend);

         kerosene.safeApprove(address(vaultManager), amountToSend);
```

NOTE: the modification can occur to `_syncClaimableAmount` itself. But Sync functions are preferred to not have any revert statement*

Sponsor: Fixed at PR-117,
commit: 854988d70aa7e829bb2a0e0a23c403c9a6fa6ca6

Al-Qa'qa': Verified. The issue has been fixed as recommended.

4.4.5 Remove declared variable that are not used

context: AtomicSwapExtension.sol#L61

Description
In `AtomicSwapExtension::afterWithdraw()`, the variable `numberOfSlots` is declared, but it is never used by the function.

AtomicSwapExtension.sol#L61

```
    function afterWithdraw(uint256 id, address vault, uint256 amount, >> address
to) external {
        require(msg.sender == address(vaultManager), OnlyVaultManager());

>>      uint256 numberOfSlots;
        ...
    }
```

The idea is the same for the Fourth parameter in `afterWithdraw`, which is not used.

Recommendations
Remove `numberOfSlots` and the fourth parameter name (not all the parameter).

```
diff --git a/src/periphery/AtomicSwapExtension.sol
b/src/periphery/AtomicSwapExtension.sol
index 6e3ade1..182f97e 100644
--- a/src/periphery/AtomicSwapExtension.sol
+++ b/src/periphery/AtomicSwapExtension.sol
@@ -55,10 +55,9 @@ contract AtomicSwapExtension is IExtension, IAfterWithdrawHook
{
         vaultManager.withdraw(noteId, fromVault, fromAmount, address(this));
     }

-    function afterWithdraw(uint256 id, address vault, uint256 amount, address to)
external {
+    function afterWithdraw(uint256 id, address vault, uint256 amount, address /*
to */) external {
         require(msg.sender == address(vaultManager), OnlyVaultManager());

-        uint256 numberOfSlots;
         address toVault;
         uint256 toAmount;
         bytes memory swapData;
```

Sponsor: Fixed at PR-122,
commit: ab0ab58b9582100aef8f18e9f0d990da88a7dfe2

Al-Qa'qa': Verified. The issue has been fixed as recommended.

4.4.6 remove useless imports from the Codebase

context:

- DyadLPStakingFactory.sol#L3
- DyadLPStaking.sol#L8-L9
- AtomicSwapExtension.sol#L8

Description
The codebase imports some functions and libraries that are not used by the files they are imported in, which makes the codebase gets larger and hard readable without reason.

DyadLPStakingFactory.sol#L3

```
import {Ownable} from "solady/auth/Ownable.sol";
import {DyadLPStaking} from "./DyadLPStaking.sol";
import {EnumerableSetLib} from "solady/utils/EnumerableSetLib.sol"; // useless
import
import {IERC20} from "forge-std/interfaces/IERC20.sol";
...
```

DyadLPStaking.sol#L8-L9

```
...
import {OwnableRoles} from "solady/auth/OwnableRoles.sol";
import {MerkleProofLib} from "solady/utils/MerkleProofLib.sol"; // useless import
import {FixedPointMathLib} from "solady/utils/FixedPointMathLib.sol"; // useless
import

contract DyadLPStaking is OwnableRoles {
```

AtomicSwapExtension.sol#L8

```
...
import {DyadHooks} from "../core/DyadHooks.sol";
import {console2} from "forge-std/console2.sol"; // useless import
import {ERC20} from "solmate/tokens/ERC20.sol";
...
```

This will result in contracts getting bigger which affects readability besides it makes SLOC increase with no need which increases the cost of the auditing process.

Recommendations
Remove useless imports from the contracts. For imports of Libraries, you need to remove the use of the library.

Sponsor: Fixed at PR-119,
commit: 397c7f100dc5c6a29779f955a6d14d55f5cc99fc

Al-Qa'qa': Verified. The issue has been fixed as recommended.