Dyad Audit Report

Version 1.0

0xnightswatch

September 29, 2025

Prepared by: 0xnightswatch

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

0xnightswatch makes 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 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

Impact

Audit Details

The findings in this document corresponds to the following commit hash:

```

https://github.com/code-423n4/2024-04-dyad.git
344a58bb84b24d6c281be038a5dff1631426e9c0
```

Scope

```

./src/core/VaultManagerV2.sol
./src/core/Vault.kerosine.bounded.sol
./src/core/Vault.kerosine.sol
./src/core/Vault.kerosine.unbounded.sol
./src/core/KerosineManager.sol
./script/deploy/Deploy.V2.s.sol
./src/staking/KerosineDenominator.sol
```

Roles

DYAD Multisig: 0xDeD796De6a14E255487191963dEe436c45995813 Description: Ability to: License new Vault Manager, License new Vaults, Change the kerosene denominator contract, Add new vaults to the Kerosene Manager

Executive Summary

- Start Date: September 23, 2025 6:00 PM
- End Date: Septmeber 28, 2025 8:30 PM

Issues found

Severity	Number of Issues
High	3
Medium	3
Low	0
Informantional	1
Total	7

Findings

In order to run the POC for each finding, create a test file in `/test/`, and then copy and paste each POC in this file and run it

```

forge test --mt {POC name} -vvv
```

```solidity
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity ^0.8.0;

import {DNft} from "../src/core/DNft.sol";
import {Dyad} from "../src/core/Dyad.sol";
import {USDCMock} from "./USCDMock.sol";
import {ERC20Mock} from "./ERC20Mock.sol";
import {OracleMock} from "./OracleMock.sol";
import {VaultManagerV2} from "../src/core/VaultManagerV2.sol";
import {KerosineManager} from "../src/core/KerosineManager.sol";
import {Licenser} from "../src/core/Licenser.sol";
import {Vault} from "../src/core/Vault.sol";
import {BoundedKerosineVault} from "../src/core/Vault.kerosine.bounded.sol";
import {UnboundedKerosineVault} from "../src/core/Vault.kerosine.unbounded.sol";
import {IAggregatorV3} from "../src/interfaces/IAggregatorV3.sol";
import {KerosineDenominatorMock} from "./KersosineDenominatorMock.sol";
import {KerosineDenominator} from "../src/staking/KerosineDenominator.sol";
import {Kerosine} from "../src/staking/Kerosine.sol";

import "forge-std/Test.sol";
import "forge-std/console2.sol";

contract POC is Test {
    address userA = makeAddr("userA");
    address userB = makeAddr("userB");
    address owner = address(this);
    address attacker = makeAddr("attacker");

    Dyad dyad;
    DNft dNft;
    Kerosine kerosine;
    USDCMock usdc;
    ERC20Mock weth;

    OracleMock usdcOracle;
    OracleMock wethOracle;
    KerosineDenominatorMock denominator;
    VaultManagerV2 manager;
    KerosineManager kerosineManager;
    Licenser licenser;
    Vault regularVaultUSDC;
    Vault regularVaultWETH;
    BoundedKerosineVault boundedKerosineVault;
    UnboundedKerosineVault unboundedKerosineVault;

    uint dNftA;
    uint dNftB;

    function setUp() public {
        licenser = new Licenser();
        kerosineManager = new KerosineManager();
        dyad = new Dyad(licenser);
        dNft = new DNft();
        kerosine = new Kerosine();
        denominator = new KerosineDenominatorMock(kerosine, address(this));
        manager = new VaultManagerV2(dNft, dyad, licenser);
        usdc = new USDCMock("USDC", "USDC");
        weth = new ERC20Mock("WETH", "WETH");
        usdcOracle = new OracleMock(1 * 1e8);
        wethOracle = new OracleMock(3_000 * 1e8);
        regularVaultUSDC = new Vault(
            manager,
            usdc,
            IAggregatorV3(address(usdcOracle))
        );
        regularVaultWETH = new Vault(
            manager,
            weth,
            IAggregatorV3(address(wethOracle))
        );
        unboundedKerosineVault = new UnboundedKerosineVault(
            manager,
            kerosine,
            dyad,
            kerosineManager
        );
        unboundedKerosineVault.setDenominator(
            KerosineDenominator(address(denominator))
        );

        boundedKerosineVault = new BoundedKerosineVault(
            manager,
            kerosine,
            kerosineManager
        );
        boundedKerosineVault.setUnboundedKerosineVault(unboundedKerosineVault);

        manager.setKeroseneManager(kerosineManager);

        licenseAllVaults();
        dNftA = dNft.mintNft(userA);
        dNftB = dNft.mintNft(userB);
    }

    function licenseAllVaults() internal {
        licenser.add(address(manager));

        kerosineManager.add(address(regularVaultUSDC));
        kerosineManager.add(address(regularVaultWETH));
        kerosineManager.add(address(boundedKerosineVault));
        kerosineManager.add(address(unboundedKerosineVault));

        licenser.add(address(regularVaultUSDC));
        licenser.add(address(regularVaultWETH));
        licenser.add(address(unboundedKerosineVault));
        licenser.add(address(boundedKerosineVault));
    }
}
```

High

[H-1] `VaultManagerV2::liquidate` Permits Users to Liquidate Their Own Positions

Description `VaultManagerV2::liquidate` does not verify that the liquidator is different from the owner of the position being liquidated.This omission allows a user to liquidate their own position when the collateralization ratio falls below 150%, which may bypassintended liquidation restrictions.

`src/core/VaultManagerV2.sol#L200-L203`

Impact

Severity: High - because an undercollateralized user can profit instead of being penalized and breaks protocol logic. Likelihood: High - any positon can become liquidatable

Proof of Concepts

```
function test_userCanLiquidateHimself() public {
        // userA got minted 160 USDC tokens which are equivelant to $160 at the currant price of $1 per USDC
        // then price is changed to make this positon liquidatble and then user liquidate himself

        uint256 userInitialUSDCBalance = 160e6;
        usdc.mint(userA, userInitialUSDCBalance);

        vm.startPrank(userA);
        manager.add(dNftA, address(regularVaultUSDC));
        usdc.approve(address(manager), 160e6);
        manager.deposit(dNftA, address(regularVaultUSDC), 160e6);
        manager.mintDyad(dNftA, 100e18, userA);
        vm.stopPrank();

        vm.roll(block.number + 1);

        // now the price dropped to 80 cents and userA become liquidatable
        // this create a bad debt to the protocol and the user to be liquidate
        // retrain all his assets
        usdcOracle.setPrice(8 * 1e7);

        vm.prank(userA);
        manager.liquidate(dNftA, dNftA);
        vm.startPrank(userA);
        manager.withdraw(dNftA, address(regularVaultUSDC), 160e6, userA);

        uint256 userFinalUSDCBalance = usdc.balanceOf(address(userA));

        assertEq(userInitialUSDCBalance, userFinalUSDCBalance);
    }
```

Recommended mitigation In `VaultManagerV2::liquidate()` add just the following check

```
function liquidate(
    uint id,
    uint to
  )
    external
      isValidDNft(id)
      isValidDNft(to)
    {
+        require(id != to, "self-liquidation");
        ....
    }
```

[H-2] Collateral Double-Counting When Vault Added to Both VaultManager and KerosineManager

Description

Regular vaults can be added to both `VaultManager` and `KerosineManager` via `add()` and `addKerosine()`. If a position deposits into a vault that exists in both managers, the same collateral is counted twice when computing total USD value. This results in an inflated collateral balance, allowing a user to appear overcollateralized beyond their actual deposited assets.

The PoC demonstrates this: a single deposit of `160 USDC` is counted as`320 USDC` in total value after adding the same vault to both managers.

Root Cause: Regular vaults need to be added to `KerosineManager` for *kerosine price calcculation*, but this allow user to add any regular vault twice.

`script/deploy/Deploy.V2.s.sol#L64-L65` `script/deploy/Deploy.V2.s.sol#L93-L96`

Impact

Severity: High — The issue directly affects the integrity of collateral accounting, which is critical for financial safety by inflating user collateral and inflating their positon, undermining liquidation logic and exposing the system for undercollaterized debt. Likelihood: High — As per the Deploy Script and protocol architecture, the protocol need to add regular vaults to both manangers `assetPrice()` in `UnboundedKerosineVault` to fucntion.

Proof of Concepts

```
function test_ifRegularVaultIsAddedToVaultLicenserAndKerosineManagerThenCollateralIsDuplicated()
        public
    {
        uint256 depositedAmount = 160e6;
        usdc.mint(userA, depositedAmount);

        vm.startPrank(userA);
        manager.add(dNftA, address(regularVaultUSDC));
        manager.addKerosene(dNftA, address(regularVaultUSDC));
        usdc.approve(address(manager), depositedAmount);
        manager.deposit(dNftA, address(regularVaultUSDC), depositedAmount);
        vm.stopPrank();

        // Assert that total USD value is doubled due to vault being counted twice
        assertEq(manager.getTotalUsdValue(dNftA), 2 * depositedAmount * 1e12);
    }
```

Recommended mitigation

The current collateral tracking across multiple managers is prone to double-counting and requires a comprehensive refactor. We recommend reviewing the architecture to ensure each vaults collateral is counted exactly once per position. If immediate refactoring is not feasible, consider implementing a check to prevent the same vault from being added to multiple managers for the same position.

[H-3] Users Cannot Remove Bounded KerosineVault After Addition

Description

Once a `boundedKerosineVault` is added to a position, it cannot be removed, permanently binding the vault to that position. While the **maximum** number of bounded and non-bounded vaults a user can add is limited to **five**, there is no mechanism to remove a bounded vault once added, except via liquidation or other extreme measures. This design reduces flexibility in collateral management.

For example, if a user has added five bounded `KerosineVaults` to their position and then attempts to liquidate another position where the Kerosine vaults differ, the user cannot remove any of their bounded vaults nor add the liquidated bounded Kerosine vault to their position to count toward their collateral.

The root cause is that the protocol requires`id2Asset[id] > 0` for a vault to be considered removable. Since bounded KerosineVaults cannot be withdrawn, this value never reaches zero, preventing removal or reassignment of the vault.

```
  function removeKerosene(
      uint    id,
      address vault
  )
    external
      isDNftOwner(id)
  {
@>  if (Vault(vault).id2asset(id) > 0)     revert VaultHasAssets();
    if (!vaultsKerosene[id].remove(vault)) revert VaultNotAdded();
    emit Removed(id, vault);
  }
```

`src/core/VaultManagerV2.sol#L113`

Impact

Severity: High — because users are permanently bound to bounded KerosineVaults, which can lock collateral, limit capital efficiency, and disrupt protocol operations. Likelihood: High — because this behavior occurs any time a bounded KerosineVault is added and cannot be removed, making it reproducible under normal usage.

Proof of Concepts

```
function test_ifUserAddBoundedKerosineVaultItCannotBeRemoved() public {
    uint256 depositAmount = 100e18;

    // Transfer KEROSINE tokens to the user
    kerosine.transfer(userA, depositAmount);

    // User deposits KEROSINE into a bounded KerosineVault
    vm.startPrank(userA);
    manager.addKerosene(dNftA, address(boundedKerosineVault)); // Add the bounded vault
    kerosine.approve(address(manager), depositAmount);          // Approve tokens for deposit
    manager.deposit(dNftA, address(boundedKerosineVault), depositAmount); // Deposit tokens
    vm.stopPrank();

    // Attempting to remove the bounded vault should revert
    vm.expectRevert();
    vm.prank(userA);
    manager.removeKerosene(dNftA, address(boundedKerosineVault));
}
```

Recommended mitigation

Introduce a mechanism for positions in boundedKerosineVault to allow the position owner to either:

- Burn the deposited KEROSINE — this can be used to influence or increase the effective asset value within the protocol.
- Transfer back to the vault owner or multisig — enabling the protocol owner to reclaim tokens from positions if necessary.

This provides flexibility and mitigates the issue of permanently locked bounded KerosineVaults.

Medium

[M-1] Return Oracle price lack sanity check for improper return values

Description `VaultManagerV2` relies on the price oracle to determine position collateralization. The contract does not perform sanity checks on the oracles return value. If the oracle returns 0, any position becomes immediately liquidatable, regardless of its actual collateral ratio. This can be exploited by a malicious oracle or misconfigured feed to trigger undesired liquidations.

When vault is called to get price from oracle, `Vault` must perform a sanity check and revert if `asnwer < 0`

`src/core/Vault.sol#L91-L103`

Impact

Severity: High — because the consequence is blanket, protocol-wide financial loss and integrity failure. Likelihood: Low — because Chainlink mainnet feeds are highly unlikely to return 0 under normal operation.

Proof of Concepts

Using the provided test test_ifPriceOracleReturnZeroOvercollateralizedUserMayBeLiquidated() — mint USDC to two users, deposit and mint DYAD so both users are overcollateralized, then force the oracle to return 0 (usdcOracle.setPrice(0)). Calling manager.liquidate(dNftA, dNftB) succeeds and treats every position as undercollateralized, allowing userB to liquidate userA despite userA being properly collateralized. The PoC demonstrates an oracle value of 0 directly breaks collateral checks and enables mass/external liquidations.

Recommended mitigation

```
  function assetPrice()
    public
    view
    returns (uint) {
      (
        ,
        int256 answer,
        ,
        uint256 updatedAt,
      ) = oracle.latestRoundData();
      if (block.timestamp > updatedAt + STALE_DATA_TIMEOUT) revert StaleData();
+     require(answer > 0, "Oracle returned zero");
      return answer.toUint256();
  }
```

[M-2] Calling `vault.oracle()` on `KerosineVault` Reverts Due to Missing Function

Description KerosineVault contracts do not implement the `.oracle()` function. Any attempt to call `vault.oracle()` on these vaults reverts with "unrecognized function selector". The `assetPrice()` function in UnboundedKerosineVault iterates over all associated vaults, calling `.oracle()` on each. Since non-regular (Kerosine) vaults do not implement `.oracle()`, this call fails, causing the function to revert and breaking price retrieval for KEROSINE positions.

`src/core/Vault.kerosine.unbounded.sol#L50-L68`

Impact

Severity: Medium — no direct fund loss, but usability and protocol reliability are affected. Likelihood: Medium — occurs whenever code iterates over vaults and assumes `.oracle()` exists.

Proof of Concepts

```
function test_getKerosinePrice() public {
       // Transfer KEROSINE to the user
       kerosine.transfer(userA, 100e18);

       // Deposit KEROSINE into UnboundedKerosineVault
       vm.startPrank(userA);
       manager.addKerosene(dNftA, address(unboundedKerosineVault));
       kerosine.approve(address(manager), 100e18);
       manager.deposit(dNftA, address(unboundedKerosineVault), 100e18);
       vm.stopPrank();

       // Retrieve asset price
       unboundedKerosineVault.assetPrice();
   }
```

Recommended mitigation

The current protocol architecture does not properly separate vault types, leading to fundamental incompatibilities (e.g., KerosineVaults not supporting .oracle()). Addressing this issue will require a comprehensive refactor of the vault management system and cannot be resolved with a minor fix. We recommend that the development team reevaluate and redesign the protocol architecture to ensure consistent and safe handling of all vault types.

[M-3] Collateral Ratio Miscalculation When Liquidating Positions with Unadded Vaults

Description When a liquidator interacts with a position that contains vaults the liquidator has not added to their own position, the liquidators collateral ratio (CR) is calculated without considering collateral from these unadded vaults. If the liquidator has already reached the maximum number of vaults allowed, they cannot claim new collateral unless they exit their current position. This can lead to the liquidator being under-collateralized relative to actual holdings, limiting their ability to mint DYAD or perform other protocol operations.

Impact

Severity: Medium-High — because the liquidators collateral ratio is underestimated, blocking valid minting and restricting capital efficiency. While it does not immediately lead to loss of funds, it creates unfair conditions and systemic inefficiencies. Likelihood: Medium — because mismatched vault participation between liquidators and liquidated positions is a realistic and expected scenario in practice.

Proof of Concepts

```
function test_ifLiquidatorLiquidateAPositionWithUnaddedVaultTheLiquidatorCrIsComputedLessThanActual()
        public
    {
        // Setup: normalize WETH price = $1 (same as USDC) to simplify calculations
        wethOracle.setPrice(1 * 1e8);

        // ----------------------
        // User A setup
        // ----------------------
        // User A deposits $160 of USDC and mints 100 DYAD
        usdc.mint(userA, 160e6);
        vm.startPrank(userA);
        manager.add(dNftA, address(regularVaultUSDC));
        usdc.approve(address(manager), 160e6);
        manager.deposit(dNftA, address(regularVaultUSDC), 160e6);
        manager.mintDyad(dNftA, 100e18, userA);
        vm.stopPrank();

        // ----------------------
        // User B setup
        // ----------------------
        // User B deposits 310 WETH (priced at $1  $310) and mints 200 DYAD
        weth.mint(userB, 310e18);
        vm.startPrank(userB);
        manager.add(dNftB, address(regularVaultWETH));
        weth.approve(address(manager), 310e18);
        manager.deposit(dNftB, address(regularVaultWETH), 310e18);
        manager.mintDyad(dNftB, 200e18, userB);
        vm.stopPrank();

        // ----------------------
        // Liquidation scenario
        // ----------------------
        // USDC price drops to $0.9  User As $160 becomes $144  CR = 1.44  liquidatable
        usdcOracle.setPrice(0.9 * 1e8);
        console2.log("Collateral Ratio of UserA: ", manager.collatRatio(dNftA));

        // User B liquidates User A by paying 100 DYAD, receiving ~75% of $144 = ~$108.8
        // Expected: User Bs total USD collateral = $310 (original) + $108.8 = ~$418.8
        uint256 UsdUserBValueBefore = manager.getTotalUsdValue(dNftB);

        vm.startPrank(userB);
        manager.liquidate(dNftA, dNftB);
        vm.stopPrank();

        // ----------------------
        // Observed issue
        // ----------------------
        // Actual: manager.getTotalUsdValue(dNftB) does NOT include the received USDC collateral
        //  Stays at $310 instead of ~$418.8
        uint256 UsdUserBValueAfter = manager.getTotalUsdValue(dNftB);
        assertEq(UsdUserBValueAfter, UsdUserBValueBefore);

        console2.log("Collateral Ratio of UserB: ", manager.collatRatio(dNftB));

        // ----------------------
        // Downstream impact
        // ----------------------
        // Because the $108.8 is ignored, CR is underestimated and minting capacity reduced.
        // Expected: User B can mint an additional ~72 DYAD safely.
        // Actual: minting reverts, as if collateral never increased.
        vm.expectRevert();
        vm.prank(userB);
        manager.mintDyad(dNftB, 72e18, userB); // <-- this should not revert
    }
```

Recommended mitigation

The liquidation flow should ensure that collateral transferred from a liquidated position is properly recognized in the liquidators accounting, even if it originates from a vault not previously added by the liquidator.

Like:

- Automatically register the vault to the liquidator when collateral is received.

Informational

[I-1] Functions `getVaults` and `hasVault` Return Only Regular (Non-Kerosine) Vaults

Description

The functions `VaultManagerV2::getVaults(uint id)` and `VaultManagerV2::hasVault(uint id, address vault)` currently operate only on regular (non-Kerosine) vaults. There is no handling for KerosineVaults, which may cause confusion for users or integrators who expect these functions to reflect all vault types.

`src/core/VaultManagerV2.sol#L290-L307`

Impact

Informational: No direct financial risk, but may lead to misunderstanding of vault composition or incomplete data when interacting with KerosineVaults. Front-end interfaces or other integrations might incorrectly assume these functions include KerosineVaults, resulting in inaccurate position or collateral displays.

Recommended mitigation

Clearly document that these functions return regular (non-Kerosine) vaults only. Optionally, add separate functions or a combined function that can return or check KerosineVaults as well.

Mocks

Please add the following two mocks contracts to the `./test/`

/2024-04-dyad/test/KersosineDenominatorMock.sol

```
// SPDX-License-Identifier: MIT
pragma solidity =0.8.17;

// import {Parameters} from "../params/Parameters.sol";
import {Kerosine} from "../src/staking/Kerosine.sol";

contract KerosineDenominatorMock {
    Kerosine public kerosine;
    address localOwner;

    constructor(Kerosine _kerosine, address _localOwner) {
        kerosine = _kerosine;
        localOwner = _localOwner;
    }

    function denominator() external view returns (uint) {
        // @dev: We subtract all the Kerosene in the multi-sig.
        //       We are aware that this is not a great solution. That is
        //       why we can switch out Denominator contracts.
        return kerosine.totalSupply() - kerosine.balanceOf(localOwner);
    }
}
```

and

/2024-04-dyad/test/USCDMock.sol

```

// SPDX-License-Identifier: MIT
pragma solidity =0.8.17;

import {ERC20} from "@solmate/src/tokens/ERC20.sol";
import {SafeTransferLib} from "@solmate/src/utils/SafeTransferLib.sol";

contract USDCMock is ERC20 {
    using SafeTransferLib for address;

    constructor(
        string memory name,
        string memory symbol
    ) ERC20(name, symbol, 6) {}

    event Deposit(address indexed from, uint256 amount);

    event Withdrawal(address indexed to, uint256 amount);

    function deposit() public payable virtual {
        _mint(msg.sender, msg.value);

        emit Deposit(msg.sender, msg.value);
    }

    function withdraw(uint256 amount) public virtual {
        _burn(msg.sender, amount);

        emit Withdrawal(msg.sender, amount);

        msg.sender.safeTransferETH(amount);
    }

    function mint(address to, uint256 amount) external virtual {
        _mint(to, amount);
    }

    receive() external payable virtual {
        deposit();
    }
}
```