Dyad

Security Review

Ethan Bennett Darklinear Solutions

Contents

About Darklinear Solutions

Darklinear Solutions leverages years of expertise in smart contract and software engineering to provide unrivaled security reviews for blockchain applications. Learn more at darklinear.com.

Introduction

Dyad is a debt-based stablecoin, pegged to the US Dollar and deployed on Ethereum mainnet. It implements a novel form of subsidized collateral in the form of its token, Kerosene. This review consists of issues discovered during the course of Code4rena's competitive audit for Dyad in April 2024. It does not represent a full and exhaustive audit of the protocol.

The findings described below are classified according to Code4rena's standards.

_

Findings Summary

Classification	Finding
High risk (H-1)	DNFT holders can inflate their collateral ratios and mint DYAD against kerosene by adding vaults to the wrong vault array
High risk (H-2)	Kerosene deposited into unbounded kerosene vault can never be withdrawn
High risk (H-3)	Vaults subsidizing collateral with kerosene may not have enough exogenous collateral to incentivize liquidation
Medium risk (M-1)	`ExoCollat` requirement can be violated
Low risk (L-1)	Kerosene denominator can be manipulated

Findings

H-1: DNFT holders can inflate their collateral ratios and mint DYAD against kerosene by adding vaults to the wrong vault array

Impact:

- There are two arrays of vaults defined in `VaultManagerV2`: `vaults` and `vaultsKerosene`. `vaults` is meant to include the vaults for exogenous collateral (WETH and wstETH), and is used to `getNonKeroseneValue`. `vaultsKerosene` is meant to include the vaults for bounded and unbounded kerosene, and is used to `getKeroseneValue`
- Since the unbounded kerosene vault is licensed like the WETH and wstETH vaults in the deploy script, it can be erroneously added to the exogenous `vaults` array using the `add` function
- Since the WETH and wstETH vaults are licensed by the `KeroseneManager` in the deploy script, they can be erroneously added to the `vaultsKerosene` array using the function `addKerosene`
- Since the bounded and unbounded kerosene vaults are not meant to be licensed by the `KeroseneManager`, they cannot be added to the `vaultsKerosene` array as intended
- Adding the exogenous collateral vaults to both arrays will cause the collateral to be counted twice in the calculation of the collateral ratio, allowing users to artificially prevent their own liquidations
- Adding the unbounded kerosene vault to the `vaults` array will cause its value to be included in the position's `NonKeroseneValue`, which will allow for DYAD to be minted against it. This allows malicious users to mint DYAD against kerosene (which undermines the value of both)

Proof of Concept:

- **Impact #1**: The bounded and unbounded kerosene vaults are meant to be added to the `vaultsKerosene` array via `VaultManagerV2.addKerosene`, but they are not meant to be licensed by the `KeroseneManager` (confirmed by the team). This is because the `KeroseneManager` licensing is meant to track the vaults contributing to the TVL in kerosene's `assetPrice` calculation — i.e., all the vaults except the ones that are meant to be added to the `vaultsKerosene` array in `VaultManagerV2`. So, when `addKerosene` checks the following, it excludes the vaults it is meant to include, and includes the vaults it is meant to exclude:

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

- **Impact #2**: Unlike the bounded and unbounded kerosene vaults, the WETH and wstETH vaults are licensed by the `KeroseneManager` in the deploy script, so they can be added to `vaultsKerosene` via `addKerosene`. By providing the same WETH or wstETH vault to both `add` and `addKerosene`, the value of the position's collateral in that vault will be counted twice when the total USD value of the DNFT is calculated: once when `getNonKeroseneValue` calculates the USD value of the collateral in each vault in the `vaults` array, and again when `getKeroseneValue` calculates the USD value of the collateral in each vault in the `vaultsKerosene` array. This inflates the position's `collatRatio`, and can be used maliciously to avoid liquidation:

```
function getTotalUsdValue(
  uint id
)
  public
  view
  returns (uint) {
      return getNonKeroseneValue(id) + getKeroseneValue(id);
}
```

- **Impact #3**: The unbounded kerosene vault is licensed by the `Licenser` along with the WETH and wstETH vaults in the deploy script.

  Since `vaultLicenser.isLicensed(unboundedKeroseneVault)` is `true`, the kerosene vault can be added to the normal `vaults` array and treated like normal collateral. This is catastrophic, because it allows users to bypass the `NotEnoughExoCollat` requirement and mintDyad against kerosene instead:

```
function mintDyad(
    uint    id,
    uint    amount,
    address to
)
    external
    isDNftOwner(id)
{
    // getNonKeroseneValue iterates over `vaults` array,
    // which can include unbounded kerosene vault

    uint newDyadMinted = dyad.mintedDyad(address(this), id) + amount;
    if (getNonKeroseneValue(id) < newDyadMinted)
        revert NotEnoughExoCollat();
    dyad.mint(id, to, amount);

    // ...
}
```

Recommended Mitigation Steps:

1. The licenser check in `addKerosene` should be changed from:
   ```
   if (!keroseneManager.isLicensed(vault))
   ```
   to:
   ```
   if (!vaultLicenser.isLicensed(vault))
   ```
2. `address public immutable kerosene` should be defined in state and set in the constructor
3. `addKerosene` should require that `vault.asset() == kerosene`
4. `add` should require that `vault.asset() != kerosene`

H-2: Kerosene deposited into unbounded kerosene vault can never be withdrawn

Impact:

- `VaultManagerV2.withdraw` calculates the USD value of the amount of collateral that the user is attempting to withdraw. It does this to ensure that the exogenous collateral value does not fall below the value of the user's `mintedDyad`
- In order to calculate the USD value, the function calls `_vault.oracle().decimals()`
- This will revert when the vault is for unbounded kerosene, since kerosene vaults do not have an oracle (or an `oracle()`)
- Since no such check exists for `deposit`, users will be able to deposit kerosene into the unbounded vault, but will never be able to withdraw it

Proof of Concept: The standard Vault.sol sets its oracle as a public state variable in the contructor:

```
IAggregatorV3 public immutable oracle;

// ...

constructor(
    IVaultManager _vaultManager,
    ERC20         _asset,
    IAggregatorV3 _oracle
) {
    vaultManager   = _vaultManager;
    asset          = _asset;
    oracle         = _oracle;
}
```

The constructor of Vault.kerosine.sol does not:

```
constructor(
    IVaultManager   _vaultManager,
    ERC20           _asset,
    KerosineManager _kerosineManager
) {
    vaultManager    = _vaultManager;
    asset           = _asset;
    kerosineManager = _kerosineManager;
}
```

This simple test also eliminates any doubt:

```
function testOracle() public {
    // defined like it is in `withdraw`
    Vault vault = Vault(address(contracts.unboundedKerosineVault));

    vm.expectRevert();
    vault.oracle();
}
```

Recommended Mitigation Steps:

`_vault.oracle()` is only called because of the check on the `NonKeroseneAmount`, which would be unaffected by a kerosene withdrawal. All this logic can therefore be applied only on the condition that `vault.asset() !== kerosene`, like so:

```
function withdraw(
    uint    id,
    address vault,
    uint    amount,
    address to
)
    public
    isDNftOwner(id)
{
    if (idToBlockOfLastDeposit[id] == block.number)
        revert DepositedInSameBlock();
    Vault _vault = Vault(vault);

    // define the kerosene address as a state variable
    // and set it in the constructor
    if (_vault.asset() !== kerosene) {
        uint dyadMinted = dyad.mintedDyad(address(this), id);
        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();
}
```

H-3: Vaults subsidizing collateral with kerosene may not have enough exogenous collateral to incentivize liquidation

Impact:

- The 20% liquidation reward comes from the liquidated DNFT's exogenous collateral vaults (WETH and wstETH)
- Kerosene can be used to subsidize a vault's collateral ratio up to the point that a position's minted DYAD is only backed 1:1 by exogenous collateral
- If a position does not have enough exogenous collateral to fund a sufficient reward, the position may not be liquidated in a timely manner (resulting in unbacked DYAD)
- If a position utilizes kerosene to the fullest extent and maintains a 150% collateral ratio with only a 1:1 ratio of exogenous collateral to minted DYAD, and the price of the exogenous collateral falls, the liquidator would actually incur a cost by liquidating the vault
- Ultimately, this means that the incentive mechanism meant to keep the protocol in good health will not function as intended, under-collateralized positions will not be liquidated, and the DYAD minted by those positions will be unbacked. This greatly compromises DYAD's dollar peg

Proof of Concept:

- First, the following fuzz test confirms that the liquidation reward is entirely funded by the collateral belonging to the liquidated position. This is intentional, since an attempt to `move` more than the value of the position's deposits would revert, but it is still important to establish.

```
function test_liquidationCalc(
    uint256 collateralRatio, uint256 totalCollateral
) public {
    collateralRatio = bound(collateralRatio, 1e18, 1.5e18);
    vm.assume(collateralRatio < 1.5e18);
    totalCollateral = bound(totalCollateral, 0, 1000000 * 1e18);

    // calculation taken from VaultManagerV2.liquidate
    uint256 liquidationEquityShare =
        (collateralRatio - 1e18).mulWadDown(0.2e18);
    uint256 liquidationAssetShare  =
        (liquidationEquityShare + 1e18).divWadDown(collateralRatio);

    assertLe(liquidationAssetShare, 1e18);
    assertLe(
        totalCollateral.mulWadUp(liquidationAssetShare),
        totalCollateral
    );
}
```

- Having confirmed this, it can also be established that:
  1. A liquidator must always pay the entirety of a position's outstanding minted DYAD
  2. The amount of collateral the liquidator receives is always less than or equal to the total exogenous collateral deposited by the liquidated position
  3. Only exogenous collateral vaults are liquidated, not kerosene

- With these basic rules defined, imagine the following scenario:
  - A user deposits $100 worth of WETH
  - The same user deposits $50 worth of kerosene
  - The user mints 100 DYAD, setting their collateral ratio at the minimum of 150%, but only backing their DYAD debt 1:1 with WETH
  - The price of WETH falls, and the user's exogenous collateral is now worth $90
  - Liquidating the vault would now require burning $100 worth of DYAD, but would yield the liquidator only $90 worth of WETH

Recommended Mitigation Steps:

Most importantly, consider liquidating a position's kerosene deposits along with their exogenous collateral. This would allow liquidators to at least be compensated for the full cost of liquidation in most situations.

For a larger-scale approach, Dutch auctions could be implemented for liquidations, allowing for collateral to be liquidated more efficiently.

M-1: `ExoCollat` requirement can be violated

Impact:

- `VaultManagerV2` requires that, in order for a DNFT holder to mint DYAD against their position, the position must have enough value locked in exogenous collateral vaults (i.e., WETH and wstETH) to cover the full value of the minted debt
- This requirement exists to ensure that DYAD cannot be backed by kerosene alone
- Since this rule is only enforced when DYAD is minted or exogenous collateral is withdrawn, it will not always hold
- This risks unbacked DYAD, loss of faith in the peg, and issues with the calculation of the `assetPrice` of kerosene

Proof of Concept:

- The requirement described above is defined in both `mintDyad` and `withdraw`:

```
// mintDyad:
if (getNonKeroseneValue(id) < newDyadMinted)
    revert NotEnoughExoCollat();

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

- It is not, however, considered in any other context where the collateral ratio is checked. This means that, over time, this requirement will not necessarily prevent kerosene from being over-represented in terms of DYAD's total collateral composition. Consider the following scenario:
  1. A user deposits $100 worth of WETH and $50 worth of kerosene
  2. The user mints 100 DYAD
  3. The value of WETH falls, such that the collateral is now worth $90 and the position is under-collateralized

4. To return the position to good standing, the user deposits $10 worth of kerosene
5. The position is now in good health according to the protocol, despite having 10 DYAD backed solely by kerosene
6. The user cannot withdraw their WETH until `getNonKeroseneValue(id) - value < dyadMinted`, but the risks to DYAD will persist until they do so (or until the position is liquidated)

Recommended Mitigation Steps: Despite the added complexity for the end user to consider, it would be better for the protocol if positions could be liquidated both when their total collateral ratio falls below the minimum and when `getNonKeroseneValue(id) < dyadMinted`.

L-1: Kerosene denominator can be manipulated

Impact:

- The kerosene denominator, used in the calculation of the kerosene `assetPrice`, subtracts the kerosene remaining to be distributed from its total supply. It defines the former by querying the balance of the `MAINNET_OWNER`
- This makes it possible for an attacker to manipulate the `assetPrice` of kerosene, by sending kerosene to `MAINNET_OWNER` and increasing the value of the denominator. They could exploit this to force vaults into liquidatable states
- This is low-severity, because it would require a large amount of kerosene to be sent to the `MAINNET_OWNER` and lost forever. It is still worth fixing, however, because it is plausible that a vault on the brink of liquidation could have enough collateral at stake to compensate the attacker for their kerosene donation

Recommended Mitigation Steps:

Consider storing the distributed amount of kerosene as a state variable in the denominator.