gkrastenov

Smart Contract Security Review

DYAD

March 8 2024

gkrastenov.com gkrastenov gkrastenov

Table of Contents

1 About gkrastenov

Georgi Krastenov, known as gkrastenov, is an independent smart contract security researcher and former smart contract engineer at Nexo. Having conducted over 15 solo smart contract security reviews and discovered numerous vulnerabilities in various protocols, he does his best to contribute to the blockchain ecosystem and its protocols by dedicating time and effort to security research and reviews. Check his previous work here or reach out on Twitter/X or Telegram @gkrastenov.

2 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

3 About DYAD

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

4 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

4.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

4.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

5 Executive summary

Overview

Project Name	DYAD
Repository	https://github.com/DyadStablecoin/contracts
Commit hash	bb0432cc2d03db0dff3dc8c274aefc6c0973af70
Review Commit hash	f08e2592435bc8c3bb0ce4b4baef0db4e3424dc6
Documentation	https://dyadstable.notion.site/DYAD-design-outline-v6-3fa96f99425e458abbe574f67b795145
Methods	Manual review

Scope

DeployBase.Kerosine.sol
Deploy.Kerosine.Mainnet.s.sol
KerosineManager.sol
Vault.kerosine.bounded.sol
Vault.kerosine.sol
Vault.kerosine.unbounded.sol
Staking.sol

Timeline

March 6, 2024	Audit kick-off
March 8, 2024	Preliminary report
March 8, 2024	Mitigation review

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	0	0	0
Low Risk	0	0	0
Informational	2	2	0
Total	**4**	**4**	**0**

6 Findings

6.1 High risk

6.1.1 Wrongly calculating the total kerosene

Severity: *High risk*

Context: Vault.kerosine.bounded.sol#L55

Description: To calculate the total kerosene amount, the balances of `UnboundedKeroseneVault` and `BoundedKeroseneVault` are subtracted. The result will indicate how much kerosene is available.

```
// In UnboundedKerosineVault contract
function getTotalKerosine() public view override returns (uint) {
return
asset.balanceOf(address(this)) -
asset.balanceOf(address(boundedKerosineVault));
}
// BoundedKerosineVault contract
function getTotalKerosine() public view override returns (uint) {
return
asset.balanceOf(address(unboundedKerosineVault)) -
asset.balanceOf(address(this));
}
```

At the core of the project is the logic that the price of Bounded kerosene is double that of Unbounded kerosene. Unfortunately, this does not guarantee that the balance of the `BoundedKeroseneVault` will always be greater than the balance of the `UnboundedKeroseneVault` contract. If users deposit more kerosene into the `BoundedKeroseneVault`, the `getTotalKerosene()` function will revert due to arithmetic underflow. This will block the calculation of the price of the kerosene.

```
// Price calculation of kerosene:
return (tvl - dyad.totalSupply()) / getTotalKerosine();
```

Another way to encounter this problem is if a malicious user directly transfers a huge amount of kerosene to BoundedKeroseneVault to block the calculation of the kerosene price.

Recommendation: Do not subtract both balances. Instead, in both contracts, track the total deposited kerosene. Also, handle the edge case when the deposited kerosene in both contracts is equal to prevent dividing by zero.

```
+ uint256 public totalDepositAmount;
function deposit(uint id, uint amount) external onlyVaultManager {
    id2asset[id] += amount;
+  totalDepositAmount += amount;
    emit Deposit(id, amount);
}

function withdraw(uint id,address to,uint amount) external onlyVaultManager {
    id2asset[id] -= amount;
+   totalDepositAmount -= amount;
    asset.safeTransfer(to, amount);
    emit Withdraw(id, to, amount);
}
```

Resolution and Client comment: Resolved. Fixed at `d38a08d4c39ae6768ec4a623b2a51de53fa20e87` commit.

6.1.2 getUsdValue is not in the correct decimal format

Severity: *High risk*

Context: Vault.kerosine.sol#L73

Description: The function `getUsdValue` is not returning the value in the correct decimal format. Currently, the `getUsdValue` function of `Vault.wsteth` returns `4412407424040000000000`, which is $4412 in 18 decimal format. However, in the context of the kerosene vault, it should be in 26 decimals. The `assetPrice` function returns a value in 8 decimal format, and 1 kerosene is equivalent to 1e18.

For example, if `assetPrice` returns 238095237, which is $2.3 in 8 decimals, the `getUsdValue` function should return the value for 1 kerosene as 238095237 * 1e18.

Recommendation: Divide by 1e8 in the `getUsdValue` function

Resolution and Client comment: Resolved. Fixed at `f08e2592435bc8c3bb0ce4b4baef0db4e3424dc6` commit.

6.2 Informational

6.2.1 The total kerosene supply will be minted to the deployer's address

Severity: *Informational*

Context: DeployBase.Kerosine.sol#L33

Description: When the `Kerosene` contract is deployed, the total kerosene supply will be minted directly to the deployer's address. The deployer's address is expected to be a hot wallet, which will not be used after the deployment of the contract.

Recommendation: A portion of the kerosene amount should be transferred to the `Staking` contract and the remaining kerosene supply to be send to `MAINNET_OWNER`.

Resolution and Client comment: Resolved. Fixed at `922c90c3b7200e4d033bbb4d4200f7094a3fc616` commit.

6.2.2 The deployer's address has ownership rights over the UnboundedKeroseneVault and BoundedKeroseneVault contracts

Severity: *Informational*

Context: DeployBase.Kerosine.sol#L55

Description: After deploying the `BoundedKeroseneVault` and `UnboundedKeroseneVault` contracts, the owner will be the deployer's address, a hot wallet not expected to be used after deployment. For security reasons, it is better the ownership rights of these contracts to belong to the `MAINNET_OWNER` address.

If the private key of the hot wallet is compromised, the hacker can call the `setUnboundedKeroseneVault()` and `setBoundedKeroseneVault()` functions to change the address of the `UnboundedKeroseneVault` and `BoundedKeroseneVault` contracts.

Recommendation: At the end of the deployment script transfer the ownership to `MAINNET_OWNER`.

Resolution and Client comment: Resolved. Fixed at `e705bee6795c2cfe4f46b00ade5b7af796668a1d` commit.