DYAD, Kerosene Vaults Security Audit

Report Version 1.0

April 2, 2024

Conducted by **Hunter Security**:

George Hunter, Lead Security Researcher
Windhustler, Senior Security Researcher

Table of Contents

1 About Hunter Security

Hunter Security is a team of independent smart contract security researchers. Having conducted over 100 security reviews and reported tens of live smart contract security vulnerabilities protecting over $1B of TVL, our team always strives to deliver top-quality security services to DeFi protocols. For security audit inquiries, you can reach out to us on Telegram or Twitter at *@georgehntr*.

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 reveal the presence of vulnerabilities **but cannot guarantee their absence**.

3 Risk classification

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

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - involves a small loss of funds or affects a core functionality of the protocol.
- **Low** - encompasses any unexpected behavior that is non-critical.

3.2 Likelihood

- **High** - a direct attack vector; the cost is relatively low compared to the potential loss of funds.
- **Medium** - only a conditionally incentivized attack vector, with a moderate likelihood.
- **Low** - involves too many or unlikely assumptions; offers little to no incentive.

3.3 Actions required by severity level

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

4 Executive summary

The Hunter Security team was engaged by DYAD to review the DYAD Kerosene Vaults contracts during the period from March 27, 2024, to March 30, 2024.

Overview

Project Name	DYAD, Kerosene Vaults
Repository	https://github.com/DyadStablecoin/contracts
Commit hash	f320675fb7c203c8efd434023f2cc2b71804d94c
Resolution	a9425372c09dda56fd6630a4f1126a51f0a3cd9a
Methods	Manual review

Timeline

-	March 27, 2024	Audit kick-off
v0.1	March 31, 2024	Preliminary report
v1.0	April 2, 2024	Mitigation review

Scope

src/core/KerosineManager.sol
src/core/Vault.kerosine.bounded.sol
src/core/Vault.kerosine.sol
src/core/Vault.kerosine.unbounded.sol
src/staking/KerosineDenominator.sol

Issues Found

High risk	2
Medium risk	0
Low risk	1
Informational	1

5 Consultants

George Hunter - a proficient and reputable independent smart contract security researcher with over 100 security engagements contributing to the security of numerous smart contract protocols in the past 2 years. Previously held roles include Lead Smart Contract Auditor at Paladin Blockchain Security and Smart Contract Engineer at Nexo. He has audited smart contracts for clients such as LayerZero, Euler, TraderJoe, Maverick, Ambire, and other leading protocols. George's extensive experience in traditional audits and meticulous attention to detail contribute to Hunter Security's audits, ensuring comprehensive coverage and preventing vulnerabilities from slipping through.

Windhustler - an independent smart contract security researcher. Having extensive experience in developing and managing DeFi projects holding millions in TVL, he is putting his best efforts into security research & reviews. He has conducted security audits for leading protocols such as LayerZero and has scored multiple top places in audit contest for projects like TapiocaDAO. Windhustler's expertise and niche skillet in cross-chain communication protocols contributes to Hunter Security's audits by uncovering unique edge cases that are usually overlooked by most auditors.

6 System overview

The scope consists mainly of the new BoundedKerosineVault and UnboundedKerosineVault contracts that will be added within the DYAD protocol to introduce the Kerosene token. The purpose of this system is to create a token (Kerosene) which market cap is pegged to the overcollateralization value of the DYAD stablecoin which should be maintained at *>=50%* of the DYAD total supply.

The KerosineManager contract and the KerosineDenominator are helper contracts that are managed by a privileged admin role. The KerosineManager is responsible for providing a list of the available pools in the protocol which hold its TVL and are therefore used to determine the price of the Kerosene token. The KerosineDenominator simply calculates the total supply of Kerosene tokens giving 2x more value to the tokens deposited in the BoundedKerosineVault as they are locked forever and can never be withdrawn.

6.1 Codebase maturity

Code complexity - *Moderate*

The code is relatively straightforward and compact in size. However, the out-of-scope components of the DYAD protocol were only partially reviewed as part of this audit meaning that there is a number of scenarios that have not been covered due to additional logic and complexity.

Security - *Moderate*

Two critical vulnerabilities were uncovered during this audit in the Kerosene price calculation function despite the fact that the contracts in scope have already been audited. The client has stated that another audit will be conducted of the full system before deployment to ensure no vulnerabilities are missed.

Decentralization - *Weak*

The protocol is heavily centralized relying on privileged addresses to manage critical parameters such as the list of DYAD vaults and Kerosene denominator.

Testing suite - *Weak*
As mentioned above, two critical vulnerabilities were identified in the core function of the contracts in scope that could have been caught via unit testing.

Documentation - *Excellent*
The provided documentation describes well the purpose and implementation of the protocol.

Best practices - *Moderate*
The contracts are compact is size and straightforward to understand and audit. However, there are numerous restricted setter functions that introduce significant centralization risks.

6.2 Privileged actors

- Bounded vault owner - has the ability to change the address of the unbounded vault which determines the price of the bounded Kerosene.
- Unbounded vault owner - has the ability to change the KerosineDenominator contract address which determines the value of all Kerosene tokens.
- Vault manager contract - has access to the *deposit*, *withdraw* and *move* functions.
- Kerosene manager contract's owner - updates the list of vaults which determine the TVL of the DYAD protocol.

6.3 Threat model

What in the DYAD Kerosene vaults has value in the market?

- The correct price of Kerosene tokens staked in the bounded and unbounded vaults.

What are the worst-case scenarios for the DYAD Kerosene vaults?

- Withdrawing tokens from the bounded Kerosene vault.
- Abusing the Kerosene price formula via the VaultManager (i.e. flashloan attacks).
- Using a wrong set of vaults to calculate the protocol's TVL.
- Being unable to interact with the vaults (Denial-of-Service).

What are the main entry points and non-permissioned functions?

- `UnboundedKerosineVault.deposit()` & `BoundedKerosineVault.deposit()` - standard deposit function incrementing the balance of the user.
- `UnboundedKerosineVault.withdraw()` - standard withdraw function decrementing the balance of the user and sending back the held tokens.
- `BoundedKerosineVault.withdraw()` - this method reverts as users are not allowed to withdraw bounded Kerosene (which means it is locked forever thus increasing its value).
- `UnboundedKerosineVault.move()` & `BoundedKerosineVault.move()` - standard move function used upon liquidation to move funds from the liquidated account to the liquidator.
- `UnboundedKerosineVault.assetPrice()` - calculates the price of a single Kerosene token by retrieving the TVL of the protocol held by all other vaults, subtracting the DYAD token total supply and diving the result by the total Kerosene supply.
- `BoundedKerosineVault.assetPrice()` - returns the price of a Kerosene token locked in the bounded vault which is simply twice the price of Kerosene staked in the unbounded vault.

6.4 Observations

Several interesting design decisions were observed during the review of the DYAD Kerosene vaults, some of which are listed below:

- A separate contract is used for the Kerosene denominator calculation as the team may decide to update that logic in the future.
- The Kerosene price is calculated using the overcollateralization of DYAD based on its totalSupply and the TVL of the protocol calculated by retrieving the value of the reserves of all vaults. A key security consideration here is that the Kerosene vaults themselves should not be included in this list of vaults used to compute the TVL.
- The price of the Kerosene deposited in the bounded Kerosene vault is twice the price of the rest tokens since those are locked forever (not-withdrawable).

6.5 Useful resources

The following resources were used during the audit:

- *DYAD Design Outline V5*

7 Findings

7.1 High

7.1.1 Incorrect precision used for calculating Kerosene asset price

Severity: *High*

Context: Vault.kerosine.unbounded.sol#L67-L70

Description: The UnboundedKerosineVault is meant calculate the Kerosene token's *assetPrice* by:

1. Retrieving the TVL stored in each single vault of the DYAD protocol via the KerosineManager.
2. Subtracting the DYAD token's *totalSupply* in order to get the overcollateralization value.
3. Dividing the result by the total Kerosene supply to get the price per single token.

```
for (uint i = 0; i < numberOfVaults; i++) {
  Vault vault = Vault(vaults[i]);
  tvl += vault.asset().balanceOf(address(vault)) * vault.assetPrice();
}
uint denominator = kerosineDenominator.denominator();
return (tvl - dyad.totalSupply()) / denominator;
```

As can be seen on the code snippet above, the *tvl* is calculated by retrieving the balance of each single vault of its asset and multiplying it by the USD price retrieved via a Chainlink oracle.

The problem is in the precision used for calculating the *tvl* and then the price:

- Different assets have different token decimals so the *tvl* will not add values in the same precision in that case.
- The return value of *vault.assetPrice* is expected to be in 8 decimals. However, the *dyad.totalSupply* is in 18 decimals meaning that unless all vaults' assets are in 10 decimals, the precision will be wrong and the calculated price will be completely wrong.

The correct implementation should retrieve and use the decimals of both the oracles and each single asset.

Recommendation: Consider implementing the following change:

```
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;
```

Resolution: Resolved.

7.1.2 The kerosene denominator uses a 3x multiplier instead of 2x

Severity: *High*

Context: KerosineDenominator.sol#L19

Description: The KerosineDenominator contract returns the denominator used by the *UnboundedKerosineVault.assetPrice* function.

```
function denominator() external view returns (uint256) {
    uint256 boundedKerosine = boundedKerosineVault.deposits();
    return boundedKerosineVault.asset().totalSupply() + 2*boundedKerosine;
}
```

It currently aims to give a 2x value of the Kerosene tokens deposited to the bounded vault since they are locked forever and thus more valuable:

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

The problem is that the *boundedKerosineVault.deposits* are already counted as part of the Kerosene *totalSupply*. Therefore, adding them 2 more times means that in the end they are counted 3 times reducing the price of a single Kerosene token.

Recommendation: Consider implementing the following change:

```
return boundedKerosineVault.asset().totalSupply() + boundedKerosine;
```

Resolution: Resolved.

7.2 Low

7.2.1 Kerosene's price could be inflated via a donation to any of the DYAD vaults

Severity: *Low*

Context: Vault.kerosine.unbounded.sol#L67

Description: One of the parameters used for calculating the Kerosene token price is the TVL in USD of all vaults within the DYAD protocol. To determine this value, the balance of each vault of its asset is retrieved in the *UnboundedKerosineVault.assetPrice* function:

```
tvl += vault.asset().balanceOf(address(vault)) * vault.assetPrice();
```

The problem is that this happens via the token's *balanceOf* function rather than reading a storage variable that performs an internal tracking of deposits as it's done in the BoundedKerosineVault. Therefore, an adversary may donate an arbitrary amount of assets to any vault which would increase the price of Kerosene without having actual collateral provided to the protocol that can be withdrawn or liquidated.

Recommendation: The fix for this issue in non-trivial since some of the vaults used have already been deployed and a storage variable tracking the deposits internally cannot be added.

Resolution: Acknowledged.

7.3 Informational

7.3.1 Typographical mistakes, code-style suggestions and non-critical issues

Severity: *Informational*

Description: The contracts contain one or more typographical issues, code-style suggestions and non-critical issues. In an effort to keep the report size reasonable, we enumerate these below:

1. The *boundedKerosineVault* storage variable is not used in the UnboundedKerosineVault contract.
2. *KerosineVault.deposit* can be marked *external*.
3. Consider using just the interface instead of _@solmate/src/tokens/ERC20.sol_.
4. The *IDNft* import in Vault.kerosine.sol is not used.
5. Contracts' ownership could be renounced while it has key role in the protocol's management. Consider using OpenZeppelin's Ownable2Step.
6. In KerosineManager the owner may add a vault that is already added or remove one that does not exist. Consider checking the return value of the *.add* and *.remove* methods.

Recommendation: Consider fixing the above typographical issues, code-style suggestions and non-critical issues.

Resolution: Acknowledged.