DYAD Security Audit

Report Version 1.0

March 1, 2024

Conducted by **Hunter Security**:

George Hunter, Lead Security Researcher

Table of Contents

1 About Hunter Security

Hunter Security is team of independent smart contract security researchers. Having conducted over 50 security reviews and reported tens of live smart contract security vulnerabilities, our team always strives to deliver top-quality security services to DeFi protocols. For security review 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 Kerosine token and staking smart contracts during the period from February 29, 2024, to March 1, 2024.

Overview

Project Name	DYAD
Repository	https://github.com/DyadStablecoin/contracts
Commit hash	2ba4bcbf25b66ecb2cc51ca4d4ccfce6685f5f3c
Resolution	0a9eb780d31e5b87e7c736e40a99c811052d5eae
Methods	Manual review

Timeline

-	February 29, 2024	Audit kick-off
v0.1	March 1, 2024	Preliminary report
v1.0	March 1, 2024	Mitigation review

Scope

src/staking/Kerosine.sol
src/staking/Staking.sol
script/deploy/Deploy.Staking.sol

Issues Found

High risk	0
Medium risk	1
Low risk	0

5 Consultants

George Hunter - a proficient and reputable independent smart contract security researcher with over 50 solo and team 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 reviews, ensuring comprehensive coverage and preventing vulnerabilities from slipping through.

6 System overview

The scope of this security audit consists of 2 smart contract and 1 deploy script. The first contract, `Kerosine` token, is a standard ERC20 token inheriting from solmate's ERC20 contract implementation and simply mints 1B tokens to the deployer that are later distributed. The second contract, `Staking`, is a fork of the well-known Synthetix StakingRewards contract, which uses the WETH:DYAD UniswapV2 LP token as a staking token and `Kerosine` as a reward token. The deploy script is used to deploy the `Kerosine` token and set up the `Staking` contract with correct parameters and supply reward funds.

6.1 Codebase maturity

Code complexity - *Satisfactory*
Both of the contracts in scope are kept minimal and straightforward.

Security - *Excellent*
The Staking contract is forked from the well-known and battle-tested Synthetix StakingRewards contract and implements minimal changes. The Kerosine token has no custom logic and is simply inheriting from solmate's ERC20 token contract.

Decentralization - *Excellent*
There is a single address with a privileged role, the Staking contract's owner, who has no control over any users' funds, and can only update the reward distribution parameters.

Testing suite - *N/A*
The code is a fork of the Synthetix StakingRewards contract with minimal changes and no additional testing has been performed.

Documentation - *N/A*
The code is a fork of the Synthetix StakingRewards contract with minimal changes and no additional documentation is needed.

Best practices - *Excellent*
The code is clean and adheres to all best practices.

6.2 Privileged actors

- owner - The `Staking` contract implements two only-owner methods that allow the owner to update the duration of the rewards distribution as well as to increase or decrease the reward rate of Kerosine tokens per second.

6.3 Threat model

What in the DYAD Kerosine staking rewards contract has value in the market?

- The staking tokens (UniswapV2 LP tokens) deposited by the users as well as the Kerosine reward tokens they receive in exchange.

What are the worst-case scenarios for the DYAD Kerosine staking rewards contract?

- Manipulating the rewards distribution logic, i.e. stealing rewards from other users.
- Draining the contract's staking tokens or stealing other users' funds.
- Stakers not being able to withdraw their staked tokens.

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

- `Staking.stake()` - the original Synthetix staking rewards stake function, performs reward update on the caller and increases the staker balance and total supply by the deposited amount on UniswapV2 LP tokens.
- `Staking.withdraw()` - the original Synthetix staking rewards withdraw function, performs reward update on the caller and decreases the staker balance and total supply by the withdrawn amount on UniswapV2 LP tokens.
- `Staking.getReward()` - the original Synthetix staking rewards harvest function, performs reward update on the caller and withdraws any rewards tokens the user has allocated.

6.4 Observations

Several interesting design decisions were observed during the review of the DYAD Kerosine token and staking rewards smart contract, some of which are listed below:

- The Staking contract is forked from the well-known and battle-tested Synthetix StakingRewards contract and implements minimal changes.
- The recover tokens and exit (batch of withdraw + claim rewards) functionalities were removed.
- The Solidity compiler version used is 0.8.17 which is several major versions higher than 0.5.x that is used by the original implementation.

6.5 Useful resources

The following resources were used during the audit:

- *Forked code*
- *Synthetix's StakingRewards contract*

7 Findings

7.1 Medium

7.1.1 Staking contract not properly initialized

Severity: *Medium*

Context: Staking.sol#L22-L25, Staking.sol#L14-L21

Description: The `Staking` contract intializes the `rewardRate` and `duration` storage variables upon construction with the following values:

```
// Duration of rewards to be paid out (in seconds)
uint public duration = 365 days * 10; // 10 years

// Reward to be paid out per second
uint public rewardRate = 2.22 * 10**18; // 2.22 per second
```

The deploy script transfers 1M reward tokens (`Kerosine`) to the staking contract. The problem is that these steps are not sufficient to properly initialize the contract's state.

Given the current setup, users will receive no rewards upon staking and unstaking since the `finishedAt` timestamp variable has not been initalized:

```
function lastTimeRewardApplicable() public view returns (uint) {
  return _min(finishAt, block.timestamp); // @audit returns 0
}

function rewardPerToken() public view returns (uint) {
  if (totalSupply == 0) return rewardPerTokenStored;

  return // @audit will return 0 unless initilized through 'notifyRewardAmount()'
      rewardPerTokenStored +
      (rewardRate * (lastTimeRewardApplicable() - updatedAt) * 1e18) /
      totalSupply;
}
```

The correct setup of the contract should include sending the intended amount of tokens to the staking contract (i.e. 1M `Kerosine` tokens) and then calling the `notifyRewardAmount` function which will calculate the appropriate reward rate. The intended reward rate is 2.22e18 units of token per second, which means that the duration should be ~5.2 days (given that we want to initially deposit just 1M `Kerosine` tokens) either updated directly in the contract's code or changed by calling `setRewardsDuration` _before_ `notifyRewardAmount`.

Recommendation: Consider adding calls to `setRewardsDuration` and `notifyRewardAmount` in the deploy script in order to properly initialize the contract.

Resolution: Resolved. The suggested fix was implemented.