Security
Audit

January 2024

ZeroLend

Audited by
Ethan Bennett

DARKLINEAR
SOLUTIONS

Contents

About Darklinear Solutions

Darklinear Solutions provides unrivaled security for blockchain applications, from the bytecode to the browser. With years of experience in smart contract development and traditional software engineering, we find the bugs that others miss. Learn more at darklinear.com.

Introduction

ZeroLend is a lending market forked from AAVE v3 and deployed on several layer two networks. This review consists of issues discovered during the course of Cantina's competitive audit for ZeroLend in January 2024. It does not represent a full and exhaustive audit of the protocol.

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

Findings Summary

In addition to the two listed findings below, three low severity vulnerabilities were submitted. However, since Cantina only published the high and medium severity issues, the low severity findings have also been omitted from this report.

Classification	Finding
Medium risk (M-1)	`ZeroLocker.supply` can be inflated without depositing tokens
Medium risk (M-2)	User maintains control of domain when offchain payments are cancelled

Findings

M-1: ZeroLocker.supply can be inflated without depositing tokens

Severity: Medium risk

Description: ZeroLocker._depositFor underlies many of the contract's lock-related functions. On L672, _depositFor adds the provided _value to the supply (which is accessible directly or via totalSupplyWithoutDecay), but not every function that utilizes _depositFor actually increases the balance of the contract. In this scenario, supply will diverge from the real supply of the contract.

This occurs in merge, which is responsible for combining two existing locks. In order to do this, the function takes two lock IDs, deletes one of them, then utilizes _depositFor to add the first lock's value to the second one. There is no new value transferred to the contract in this process, but the value stored as supply still increases.

Impact: This can be used to grief the contract with negligible cost to the attacker and little recourse for the protocol (since supply cannot be updated, the only mitigation after launch would be upgrading to a new implementation). It is also guaranteed to happen as a side-effect of the normal usage of the merge function.

The raw supply (as opposed to the values calculated in _checkpoint) is not referenced anywhere in the incentive contracts, so the impact is limited. However, it is easy to imagine present or future features on- or off-chain that could be impacted by this value being wrong.

Proof of concept: This test demonstrates the contract's supply increasing when a user calls merge, despite the contract's balance remaining the same before and after the function executes.

Recommendation: This can be avoided by checking the depositType before increasing the supply:

```
if (depositType != DepositType.MERGE_TYPE) {
    supply = supplyBefore + _value;
}
```

M-2: Missing checks for expired locks

Severity: Medium risk

Description: ZeroLock._depositFor explicitly assumes that any locks it has been passed have not expired:

```
// _locked.end > block.timestamp (always)
```

While true in most cases, merge does not require this condition, which means that expired locks can slip into _depositFor (and subsequently _checkpoint).

Proof of concept: This test merges two expired locks and shows that _depositFor has been successfully executed with one of them.

Recommendation: All other entry points to _depositFor implement the following:

```
require(_locked.end > block.timestamp, "Cannot add to expired lock.");
```

A similar check should be added to `merge`, or the assumption should be removed from the inline documentation.