Zellic

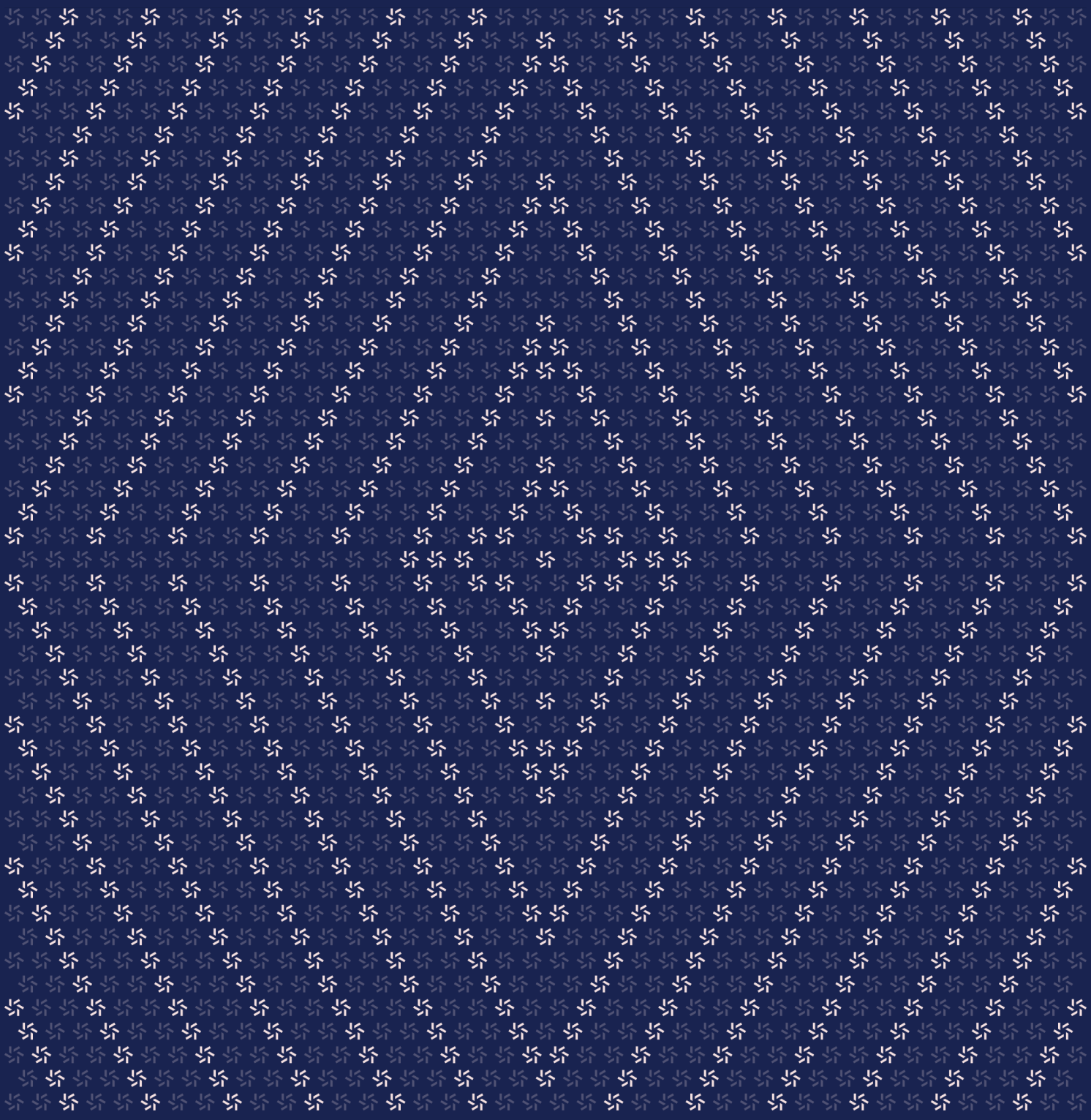
Prepared for
Toby (toby@molend.org)
Ben (ben@molend.org)
Molend Labs

Prepared by
Nipun Gupta
Jaeeu Kim
Junyi Wang
Zellic

March 5, 2024

Molend Protocol

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Molend Labs from February 29th to March 1st, 2024. During this engagement, Zellic reviewed Molend Protocol's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could potential vulnerabilities be exploited in the modified codebase, introducing risks not present in the original Aave protocol?
- Are there any changes that restrict users from performing actions that were originally intended in the Aave protocol?
- Do the modifications align with the best practices and security standards?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

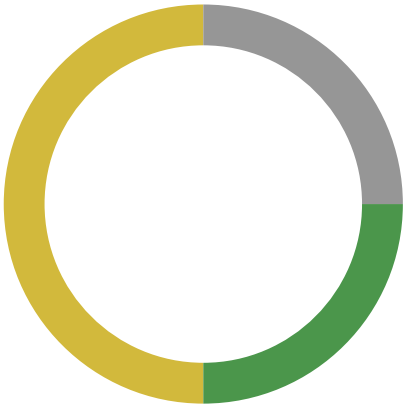
1.4. Results

During our assessment on the scoped Molend Protocol contracts, we discovered four findings. No critical issues were found. Two findings were of medium impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Molend Labs's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	2
Low	1
Informational	1



2. Introduction

2.1. About Molend Protocol

Molend Protocol is an Aave-like lending protocol on Mode blockchain, which allows users to deposit and borrow crypto assets.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

> **Basic coding mistakes.** Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.
>
> **Business logic errors.** Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.
>
> **Integration risks.** Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.
>
> **Code maturity.** We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather

than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Molend Protocol Contracts

Repository	https://github.com/molend-labs/molend-protocol ↗
Version	molend-protocol: `d8fb96ce2cd847b880f3c975d3e31e214aae3675`
Programs	• misc/UiIncentiveDataProvider.sol • utils/ChainlinkSourcesRegistry.sol • protocol/tokenization/AaveCollector.sol • misc/UiPoolDataProvider.sol
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of six person-days. The assessment was conducted over the course of two calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Nipun Gupta
Engineer
nipun@zellic.io

Jaeeu Kim
Engineer
jaeeu@zellic.io

Junyi Wang
Engineer
junyi@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

Febuary 29, 2024	Kick-off call
Febuary 29, 2024	Start of primary review period
March 1, 2024	End of primary review period

3. Detailed Findings

3.1. Owner set for implementation instead of proxy

Target	ChainlinkSourcesRegistry		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The contract ChainlinkSourcesRegistry is an upgradeable contract inheriting Ownable and VersionedInitializable abstract contracts. The Ownable contract transfers the ownership to the deployer of the contract in the constructor:

```
constructor() internal {
    address msgSender = _msgSender();
    _owner = msgSender;
    emit OwnershipTransferred(address(0), msgSender);
  }
```

This would make the deployer as the default owner of the contract. As constructors are not used in proxy-based upgradable contracts, the owner would not be correctly set.

Impact

Owner would not be set correctly, and all the `onlyOwner` functions will revert.

Recommendations

Use OwnableUpgradeable instead of Ownable and call `__Ownable_init` in the `initialize` function.

Remediation

This issue has been acknowledged by Molend Labs.

Molend Labs provided the following response:

> For ChainlinkSourcesRegistry, actually we are not using it via a proxy, instead the contract is deployed and constructed directly because the logic is quite straightforward and it's only needed by subgraphs so upgrading is unnecessary in our case.

3.2. Using deprecated Chainlink function

Target	UiPoolDataProvider		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

In the UiPoolDataProvider contract, The function `getReservesData()` is used to return the list of aggregated reserves data. This uses `latestAnswer()` to get the Chainlink oracle price.

```
function getReservesData(ILendingPoolAddressesProvider provider)
  public
  view
  override
  returns (AggregatedReserveData[] memory, BaseCurrencyInfo memory)
{
  // ...
  BaseCurrencyInfo memory baseCurrencyInfo;
  baseCurrencyInfo.networkBaseTokenPriceInUsd
    = networkBaseTokenPriceInUsdProxyAggregator
    .latestAnswer();
  baseCurrencyInfo.networkBaseTokenPriceDecimals
    = networkBaseTokenPriceInUsdProxyAggregator
    .decimals();

  try oracle.BASE_CURRENCY_UNIT() returns (uint256 baseCurrencyUnit) {
    baseCurrencyInfo.marketReferenceCurrencyUnit = baseCurrencyUnit;
    baseCurrencyInfo.marketReferenceCurrencyPriceInUsd
    = int256(baseCurrencyUnit);
  } catch (
    bytes memory /*lowLevelData*/
  ) {
    baseCurrencyInfo.marketReferenceCurrencyUnit = ETH_CURRENCY_UNIT;
    baseCurrencyInfo
      .marketReferenceCurrencyPriceInUsd
    = marketReferenceCurrencyPriceInUsdProxyAggregator
      .latestAnswer();
  }

  return (reservesData, baseCurrencyInfo);
}
```

According to Chainlink's documentation, the `latestAnswer()` function is deprecated. This function does not revert if no answer was reached and will return zero. As this function provides the last recorded value, it does not offer any additional data to verify the returned data such as update time, round, and raw price.

Impact

If the function `latestAnswer()` fails to get the price, it will return zero. In this case, the protocol that uses UiPoolDataProvider is not working as expected.

Recommendations

Use `latestRoundData` and `getRoundData` to get the price instead of `latestAnswer`. It is advised in Chainlink's documentation.

Both `latestRoundData` and `getRoundData` provide additional data to verify that the returned data is not stale or invalid.

Remediation

This issue has been acknowledged by Molend Labs.

Molend Labs provided the following response:

> We are using a pyth-chainlink adaptor to provide token prices from pyth, so the latestAnswer() function is actually pointing to a pyth oracle thus it won't be affected by chainlink's deprecation.

3.3. Using invalid Maker token address

Target	UiPoolDataProvider		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In the UiPoolDataProvider contract, the variable `MKRAddress` is used to check the address of the Maker token. But it is hardcoded.

```
contract UiPoolDataProvider is IUiPoolDataProvider {
  using WadRayMath for uint256;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using UserConfiguration for DataTypes.UserConfigurationMap;

  IChainlinkAggregator
    public immutable networkBaseTokenPriceInUsdProxyAggregator;
  IChainlinkAggregator
    public immutable marketReferenceCurrencyPriceInUsdProxyAggregator;
  uint256 public constant ETH_CURRENCY_UNIT = 1 ether;
  address public constant MKRAddress
    = 0x9f8F72aA9304c8B593d555F12eF6589cC3A579A2;

  // ...

  function getReservesData(ILendingPoolAddressesProvider provider)
    public
    view
    override
    returns (AggregatedReserveData[] memory, BaseCurrencyInfo memory)
  {
    // ...
    if (address(reserveData.underlyingAsset) == address(MKRAddress)) {
      bytes32 symbol
    = IERC20DetailedBytes(reserveData.underlyingAsset).symbol();
      reserveData.symbol = bytes32ToString(symbol);
    } // ...
```

Maker token is deployed on Ethereum Mainnet at `0x9f8F72aA9304c8B593d555F12eF6589cC3A579A2`. But in Mode blockchain, it is not deployed.

So, if this protocol is deployed on Mode blockchain, it will not work as expected.

Impact

If Maker token is deployed at another address in Mode blockchain, the function `getReservesData()` will fail with revert because `getReservesData()` could not cast Maker's symbol `bytes32` to `string`.

In this case, this protocol could not get reserves data about Maker tokens. This means the protocol could not support Maker tokens.

Recommendations

Use a Maker token address in Mode blockchain instead of an address in the Ethereum network.

Remediation

This issue has been acknowledged by Molend Labs.

Molend Labs provided the following response:

> MKR doesn't exist on Mode yet. We will keep it this way for now and update the address once it's deployed.

3.4. Using unnecessary code

Target	UiIncentiveDataProvider		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the UiIncentiveDataProvider contract, There are some try-catch statements to handle errors from calling functions. However, `bytes memory`, which is unused, is present in the catch block.

```
function _getReservesIncentivesData(ILendingPoolAddressesProvider provider)
  private
  view
  returns (AggregatedReserveIncentiveData[] memory)
{
  // ...

  for (uint256 i = 0; i < reserves.length; i++) {
    // ...

    try IStableDebtToken(baseData.aTokenAddress).getIncentivesController()
    returns (
      IAaveIncentivesController aTokenIncentiveController
    ) {
      // ...
      }
    } catch (
      bytes memory /*lowLevelData*/
    ) {
      // Will not get here
    }

  // ...
```

Recommendations

Delete the unused keyword for code maturity.

Remediation

This issue has been acknowledged by Molend Labs.

Molend Labs provided the following response:

> Since this contract is directly forked from the latest AAVE mainnet deployment, we'd prefer to keep it untouched.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Rounding issue in Aave pool

The Molend protocol is built on top of the Aave protocol. A notable concern arises due to a rounding issue present in the Aave protocol, which subsequently impacts Molend. This issue has the potential to be exploited in newly deployed markets, specifically when the aToken's `totalSupply` is zero.

In the initial stages of a market launch, an attacker could exploit a brief window to manipulate the exchange rate. This manipulation involves depositing a substantial amount of tokens into the pool and subsequently withdrawing all but one Wei of tokens. This action significantly inflates the value of the `liquidityIndex` as the `totalSupply` is then set to one Wei, leading to a discrepancy in the exchange rate. As a result, the attacker may borrow more tokens than expected.

The issue has been previously addressed and mitigated within the original Aave protocol. This mitigation involved incorporating an initial deposit requirement during the creation of new markets, ensuring that they are never left in an empty state.

4.2. Stable-rate mode borrowing vulnerability in Aave pools

A bug was recently disclosed to Aave concerning stable pools. As per Aave's recommendation, it is advised to disable stable-rate mode borrowing as a measure to mitigate the undisclosed vulnerability. More details regarding the issue could be found in Aave's security incident report ↗.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: ChainlinkSourcesRegistry.sol

Function: `updateAggregators(address[] memory assets, address[] memory aggregators)`

The function updates the `aggregatorsOfAssets` mapping.

Inputs

- `assets`
  - **Control**: Arbitrary.
  - **Constraints**: No constraints.
  - **Impact**: The mapping key to be updated.
- `aggregators`
  - **Control**: Arbitrary.
  - **Constraints**: No constraints.
  - **Impact**: The mapping value to be updated.

Branches and code coverage

Intended branches

- The function updates the `aggregatorsOfAssets` mapping for the provided input.
  - ☑ Test coverage

Negative behavior

- Revert if caller is not admin.
  - ☐ Negative test

Function call analysis

- No external function calls found.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Mode mainnet.

During our assessment on the scoped Molend Protocol contracts, we discovered four findings. No critical issues were found. Two findings were of medium impact, one was of low impact, and the remaining finding was informational in nature. Molend Labs acknowledged all findings and implemented fixes.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.