Zellic

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

Prepared by
Nipun Gupta
Kuilin Li
Zellic

April 1, 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 April 1st to April 2nd, 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:

- Is it possible for an attacker to borrow funds on behalf of other users?
- Does the looping function as expected?
- Could there be any scenario where the tokens of the users could be stuck indefinitely in the contract?
- Does the Looper safely interact with uncontrolled ERC-20 asset contracts?

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.

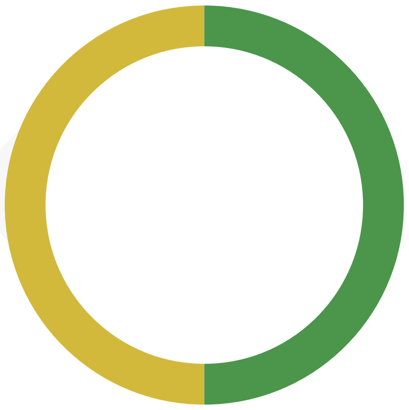
1.4. Results

During our assessment on the scoped Molend Protocol contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and one was of low impact.

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	1
🟩 Low	1
⬜ Informational	0



2. Introduction

2.1. About Molend Protocol

Molend Labs contributed the following description of Molend Protocol:

> Molend Protocol is an AAVE-like lending protocol on Mode blockchain.

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: `2861fed5383643a30568089d6fd202b6b5420994`
Program	misc/Looping.sol
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

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

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

Kuilin Li
Engineer
kuilin@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

April 1, 2024	Kick-off call
April 1, 2024	Start of primary review period
April 2, 2024	End of primary review period

3. Detailed Findings

3.1. Lending pool address may change

Target	Looping.sol		
Category	Protocol Risks	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The Looping contract saves the lending pool address as an immutable constant `LENDING_POOL` when the contract is constructed:

```
constructor(ILendingPoolAddressesProvider provider, IWETH weth) public {
  ADDRESSES_PROVIDER = provider;
  LENDING_POOL = ILendingPool(provider.getLendingPool());
  WETH = weth;
}
```

However, according to the Aave documentation ↗ for the LendingPoolAddressesProvider contract,

> Addresses register of the protocol for a particular market. This contract is immutable and the address will never change. Whenever the LendingPool contract is needed, we recommended you fetch the correct address from the LendingPoolAddressesProvider smart contract.

This means that the address of the lending pool may change.

Impact

If the address of the lending pool ever changes, the Looper will stop functioning.

Recommendations

We recommend either explicitly documenting that the address of the lending pool will never change or only immutably saving the address of the LendingPoolAddressesProvider contract, then making a call to `getLendingPool` before starting each instance of looping.

Remediation

This issue has been acknowledged by Molend Labs, and a fix was implemented in commit `94ee16d1` ↗.

3.2. Calling approve might revert for some tokens

Target	Looping.sol		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

Some tokens do not correctly implement the EIP-20 standard, and their `approve` function returns void instead of a successful boolean. Using these functions as the asset in the call to `loop` would fail in `internalLoop` as it uses a high-level call that assumes `IERC20.approve` has a bool return value.

```
function internalLoop(
  address user,
  address asset,
  uint256 amount,
  uint256 borrowRatio,
  uint256 loopCount
) internal returns (uint256, uint256) {
  if (IERC20(asset).allowance(address(this), address(LENDING_POOL)) == 0) {
      require(IERC20(asset).approve(address(LENDING_POOL), uint256(-1)),
    "Failed to approve");
  }
  //...
}
```

This would revert all the calls to `loop` if any of these tokens are used as an asset.

Impact

If the Looper is used with a token that doesn't correctly implement the EIP-20 standard, the call will unexpectedly revert due to a failed return data type check.

Recommendations

We recommend using OpenZeppelin's SafeERC20 ↗ versions with the `safeIncreaseAllowance` and `safeDecreaseAllowance` functions that handle the return-value check as well as nonstandard-compliant tokens.

Remediation

This issue has been acknowledged by Molend Labs, and a fix was implemented in commit `94ee16d1` ↗.

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. Implementing permits would improve onboarding

Currently, in order for a new user to use the Looping contract, they must set up an ERC-20 approval for the contract to spend their underlying asset and also a credit delegation so that the contract can borrow on the lending pool on their behalf.

This means that, if a new user is an EOA, they will need to submit two transactions to set things up before submitting a third transaction to actually perform the looping.

We recommend supporting ERC-20 permit transfers and Aave permit-borrow delegations (`delegationWithSig`) so that a user can submit two signatures to the Looper and execute the entire onboarding experience in one transaction.

4.2. No way to set referral code

Aave has a referral-code mechanism, which is currently disabled. However, future versions of Aave or the lending pool may enable the feature.

We recommend passing through a referral code from the user into the calls to the lending pool, instead of having the referral code be hardcoded to zero, in case a future update causes referral codes to yield economic incentives that a user would like to (or, would like their referrer to) partake in.

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: Looping.sol

Function: `loop(address asset, uint256 amount, uint256 borrowRatio, uint256 loopCount)`

The function facilitates repetitive depositing and borrowing of the specified asset within the protocol.

Inputs

- `asset`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The asset to deposit and borrow.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The amount of asset to deposit.
- `borrowRatio`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Should be less than or equal to 10,000.
    - **Impact**: The ratio of amount deposited, to be borrowed per iteration.
- `loopCount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The number of times to deposit and borrow from the pool.

Branches and code coverage (including function calls)

Intended branches

- Transfer the funds to this contract from the `msg.sender` and deposit/borrow the funds to the lending pool, `loopCount` number of times.
    - ☑ Test coverage
- The function should properly approve the assets to the lending pool if the initial allowance

is 0.
- [x] Test coverage

Negative behavior

- Revert if `borrowRatio` is greater than 10000.
  - [ ] Negative test
- Revert if the token is not supported.
  - [x] Negative test

Function call analysis

- `IERC20(asset).safeTransferFrom(user, address(this), amount)`
  - **What is controllable?** `asset`, `user`, and `amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `internalLoop -> IERC20(asset).allowance(address(this), address(LENDING_POOL))`
  - **What is controllable?** `asset`.
  - **If the return value is controllable, how is it used and how can it go wrong?** If the return value is 0, `type(uint256).max` amount of assets are approved to the lending pool.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `internalLoop -> IERC20(asset).approve(address(LENDING_POOL), uint256(-1))`
  - **What is controllable?** `asset`.
  - **If the return value is controllable, how is it used and how can it go wrong?** If the token is ERC-20 compliant, it would return true/false depending on the status of the approval; if not, the function would not return anything.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `internalLoop -> LENDING_POOL.deposit(asset, amount, user, 0)`
  - **What is controllable?** `asset`, `amount`, and `user`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `internalLoop -> LENDING_POOL.borrow(asset, amount, 2, 0, user)`
  - **What is controllable?** `asset`, `amount`, and `user`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.

Function: `loopETH(uint256 borrowRatio, uint256 loopCount)`

The function facilitates repetitive depositing and borrowing of ETH within the protocol.

Inputs

- `borrowRatio`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be less than or equal to 10,000.
  - **Impact**: The ratio of amount deposited, to be borrowed per iteration.
- `loopCount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The number of times to deposit and borrow from the pool.

Branches and code coverage (including function calls)

Intended branches

- The function should convert the provided ETH to WETH and deposit/borrow the WETH to the lending pool, `loopCount` number of times.
  - ☑ Test coverage
- The function should properly approve WETH to the lending pool if the initial allowance is 0.
  - ☑ Test coverage

Negative behavior

- Revert if `borrowRatio` is greater than 10000.
  - ☐ Negative test
- Revert if `msg.value` is 0.
  - ☐ Negative test

Function call analysis

- `WETH.deposit{value: amount}()`
  - **What is controllable?** `amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `internalLoop -> IERC20(asset).allowance(address(this), address(LENDING_POOL))`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?**

    If the return value is `0`, `type(uint256).max` amount of assets are approved to the lending pool.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `internalLoop -> IERC20(asset).approve(address(LENDING_POOL), uint256(-1))`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** The call returns true.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `internalLoop -> LENDING_POOL.deposit(asset, amount, user, 0)`
  - **What is controllable?** `amount` and `user`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `internalLoop -> LENDING_POOL.borrow(asset, amount, 2, 0, user)`
  - **What is controllable?** `amount` and `user`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.

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 two findings. No critical issues were found. One finding was of medium impact and one was of low impact.

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.