0liqlend Security Review

Reviewer
m4k2
December 10, 2024

1 Executive Summary

Over the course of 3 days in total, 0liqlend engaged with Beyond for a security review.

The review identified several design choices and implementation details that warrant attention, particularly around asset correlation and systemic risks.

Summary

Type of Project	Lending Protocol
Timeline	Nov 2, 2024 - Nov 4, 2024
Methods	Manual Review

Total Issues

High Risk	2
Medium Risk	2
Low Risk	5

The findings focus on systemic risks related to asset correlation and potential depegging scenarios, while acknowledging certain design choices that shape the protocol's behavior.

Contents

1 Executive Summary **1**

2 Beyond **3**

3 Scope of the audit **3**

4 Findings classification **4**
4.1 Likelihood and Impact matrix 4
4.2 Core issue definition 4
4.3 Protocol's design and functionality 5

5 Findings **6**
5.1 High Risk 6
5.1.1 Liquidations Only Possible After Loan Expiry, Not On LTV Breach 6
5.1.2 Price Manipulation Risk Due to Hardcoded Asset Prices 7
5.2 Medium Risk 8
5.2.1 Liquidation Mechanism Incorrectly Liquidates All Borrower Positions 8
5.2.2 Missing Collateral Top-Up Function Prevents Users from Avoiding Liquidation 8
5.3 Low Risk 9
5.3.1 Incorrect decimal handling in lending offers can lead to value discrepancies 9
5.3.2 Missing Nimbora Token Support in Asset Validation 9
5.3.3 Code Organization Can Be Improved by Consolidating Stablecoin Category Checks 10
5.3.4 Hardcoded Admin Address Reduces Protocol Flexibility and Security 11
5.3.5 ERC20 Interface Compatibility Risk Due to Custom Implementation 13

2 Beyond

Beyond is a team of Web3 security engineers unified by a common objective: to ensure the continued security of blockchain technology. Our collective expertise encompasses the full spectrum of Web3 technologies, from protocol architecture to smart contract implementation. We specialize in comprehensive security assessments across diverse blockchain environments, with expertise in Solidity and EVM-based protocols, Rust, and Cairo. Our assessments combine a rigorous methodology with practical insights, leveraging our expertise to provide thorough coverage of potential vulnerabilities while maintaining the highest standards of quality.

Disclaimer: Please note that this security review does not guarantee against a hack. It represents a snapshot in time of the code, as reviewed by m4k2. Any modifications to the code will require a new security review.

3 Scope of the audit

The audit was conducted on the following commit hash: 0ffc13976c6295cdf621ef07a262724cf6845064

```
constants.cairo
datastructures.cairo
integration.cairo
lib.cairo
mock_erc20.cairo
token_encoding.cairo
utilities.cairo
```

4 Findings classification

In order to fairly evaluate the severity of an issue, it is correlated with the following components:

- the `likelihood/impact` matrix
- the core issue definition
- the protocol's design and functionality

It is assumed that privileged parties (admins, bots...) will act for the benefit of the protocol and not deliberately perpetrate any malicious actions against it.

Moreover, it is the protocol's responsability to securely manage their accounts to prevent eventual compromissions.

Thus, restricted functions requiring privileges will be assumed to be called with legitimate values.

4.1 Likelihood and Impact matrix

The following matrix facilitates the evaluation of an issue based on its likelihood and impact.

Likelihood/Impact	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

4.2 Core issue definition

This section details the severity of an issue based upon its damage for the protocol.

It is broken down into :

- a brief introduction of the issue's damage at its core
- non-exhaustive examples of such issue

Critical

- Empty or freeze the contract's holdings with no pre-condition, very low capital investment and no time constraint.
- Disrupt the system with impossibility to recover.

  Examples:

  - Infinitely repeatable risk-free trades
  - Draining the system capital with profit
  - Continuous Denial Of Service (DoS) against the protocol's essential functionalities
  - Prevent users from withdrawing their funds or interacting with them

High

- Considerable amount of funds are lost.
- System can be disrupted with difficulty to recover.

  Examples:

  - User position can be driven towards liquidation by a malicious actor
  - Functionality is disrupted for more than an hour
  - Protocol's fundamental functionality is flawed
  - Unauthorized party can act on behalf of another user

Medium

- Funds are lost under certain conditions.
- System can be temporarily disrupted but can recover.
- Functionality is missing or incorrectly implemented.
- Non-compliance with standards meaningful for the protocol.

  Examples:

  - Lack of slippage
  - Fee calculations are flawed resulting in users/protocol losing a relatively low percentage of their funds
  - Admin-restricted transaction damages a user's position
  - Functionality is disrupted for less than an hour
  - System not compliant with EIP-712

Low

- Funds are not at risk but the handling of the state might be incorrect or not handled appropriately.

  Examples:

  - User mistakes that could have been prevent
  - Issues that may occur in a future implementation of the protocol
  - Weird token behaviors (fee-on-transfer, rebase, blacklist...) unless strong will for the protocol to integrate them
  - Incorrect event emission unless the system's functionalities heavily depend on off-chain listeners.

4.3 Protocol's design and functionality

Every protocol is designed to provide its own set of functionalities and services.

Thus, the threat model that corresponds to a particular protocol might differ from another. In this manner, the same issue can be considered damageful for a particular protocol and irrelevant for another.

This disparity is taken into account when defining the severity of an issue, making it fit the protocol's specific design.

5 Findings

5.1 High Risk

5.1.1 Liquidations Only Possible After Loan Expiry, Not On LTV Breach

Description: The liquidation mechanism in the lending protocol is flawed as it only allows liquidations to occur after a loan's maximum duration has expired, rather than when the loan-to-value (LTV) ratio falls below the required threshold.

In the `liquidate()` function, the only check performed before allowing liquidation is:

```
assert!(current_date > match_offer.date_taken + match_offer.maximal_duration, "Loan cannot be liquidated
↪  yet");
```

This means that even if the value of the collateral drops significantly, causing the LTV ratio to fall below the minimum threshold, the loan cannot be liquidated until its maximum duration expires. This creates significant risk for lenders as they have no protection against rapid collateral value depreciation during the loan term.

For example, if a borrower takes out a loan with ETH as collateral when ETH is worth $2000, and ETH drops to $1000 during the loan term, the loan would become undercollateralized. However, lenders would be unable to liquidate the position to protect themselves until the loan's maximum duration expires, leading to significant losses.

This is particularly dangerous in volatile crypto markets where asset values can change dramatically in short time periods. Most lending protocols implement immediate liquidation triggers based on LTV ratios specifically to protect against this scenario.

The chosen approach of only allowing matches between highly correlated assets (wETH/ETH, stablecoins, etc.) presents certain risks. While this strategy is effective in most cases, it exposes the protocol to significant systemic risks if one of the assets depegs. Such an event could lead to the insolvency of lenders and seriously compromise the protocol's viability. The decision to only liquidate after expiration date remains a concern.

Recommendation: The protocol should implement LTV-based liquidation triggers in addition to the existing expiry-based liquidation. This would allow positions to be liquidated when they become undercollateralized, regardless of the loan duration.

Add a function to check if a loan's current LTV ratio is below the minimum threshold:

```
fn is_undercollateralized(match: Match) -> bool {
    let current_collateral_value = get_current_collateral_value(match.collateral);
    let current_loan_value = get_current_loan_value(match);
    return current_collateral_value < minimum_required_collateral(current_loan_value);
}
```

Then modify the liquidation check in `liquidate()`:

```
assert!(
    current_date > match_offer.date_taken + match_offer.maximal_duration ||
    is_undercollateralized(match_offer),
    "Loan cannot be liquidated yet"
);
```

This would allow liquidations to occur either when the loan expires OR when the position becomes undercollateralized.

Project: Acknowledged.

Beyond: Acknowledged.

5.1.2 Price Manipulation Risk Due to Hardcoded Asset Prices

Relevant Context: The protocol allows users to lend and borrow assets using other assets as collateral. The value of collateral is determined by the `compute_value_of_asset()` function which uses hardcoded prices set by admins via `setPrice()`.

Description: The protocol uses hardcoded prices set by admins to determine collateral values rather than getting real-time prices from an oracle. This creates a significant risk as the hardcoded prices can deviate from actual market prices.

The issue stems from the `compute_value_of_asset()` function:

```
fn compute_value_of_asset(self: @ContractState, amount: u256, address: ContractAddress) -> u256 {
    let erc20 = IERC20Dispatcher { contract_address: address };
    let price = self.price_information.read(address);
    let ltv = self.ltv_information.read(address);
    aux_compute_value_of_asset(amount, erc20, price, ltv)
}
```

This function reads prices from storage that are set by admins via `setPrice()`. When these prices deviate from market prices, it creates arbitrage opportunities that can be exploited to drain value from the protocol.

Given that WETH is expressed in ETH (1:1) and each stablecoin is valued against other stablecoins, assets are expressed with the intrinsic value of their group. If an asset within a group deviates from its group's price, its price differs from its intrinsic value. This makes it possible to create arbitrage between assets within a group. While this works fine in 90% of cases, when an asset depegs or diverges from its intrinsic value, all lenders and borrowers in that group become vulnerable to attacks. For an attack to be profitable, the depeg needs to be greater than the LTV of each collateral (not the protocol's LTV, as the protocol doesn't have a global LTV), and having an LTV too close to 100% is problematic.

Impact: Price discrepancies between hardcoded and market prices can be exploited to:

1. Over-collateralize loans when hardcoded prices are higher than market
2. Under-collateralize loans when hardcoded prices are lower than market
3. Perform arbitrage trades using the protocol's mispriced assets

This leads to significant losses for lenders and protocol insolvency.

Likelihood: Crypto asset prices are highly volatile and change frequently. It's virtually guaranteed that hardcoded prices will deviate from market prices, creating exploitable opportunities.

Recommendation: Integrate a reliable price oracle system like pragma to get real-time market prices instead of using hardcoded values. The `compute_value_of_asset()` function should be modified to fetch prices from the oracle:

```
- let price = self.price_information.read(address);
+ let price = oracle.getPrice(address);
```

Additionally:

- Implement price freshness checks
- Consider using TWAPs for additional price manipulation resistance

Project: Acknowledged.

Beyond: Acknowledged.

5.2 Medium Risk

5.2.1 Liquidation Mechanism Incorrectly Liquidates All Borrower Positions

Description: The liquidation mechanism in the lending protocol has a critical flaw where liquidating a single overdue position results in liquidating all of the borrower's positions that share the same collateral, even if those positions are healthy and well-collateralized.

This occurs in the `liquidate()` function where, after identifying an overdue position to liquidate, the code loops through all matches to find any that use the same collateral from the same borrower. When found, these positions are also marked as liquidated and their collateral is distributed to lenders, regardless of whether those positions were actually eligible for liquidation.

The root cause is that the liquidation logic does not:

1. Check if each position is actually eligible for liquidation (i.e. overdue or under-collateralized)
2. Calculate the minimum amount of collateral needed to cover just the defaulted position
3. Allow partial liquidations that would leave healthy positions intact

This creates significant unnecessary losses for borrowers as healthy positions are forcefully closed and all their collateral is liquidated, even when liquidating just the problematic position would have been sufficient to make lenders whole.

Recommendation: The liquidation mechanism should be redesigned to:

1. Only liquidate positions that are actually eligible for liquidation (overdue or under-collateralized)
2. Calculate the minimum amount of collateral needed to cover:
    - The defaulted loan amount plus interest
    - Liquidation fees/penalties
    - Protocol fees
3. Implement partial liquidations where only the necessary amount of collateral is sold, preserving the remainder for other healthy positions
4. Update collateral accounting to track which portions of shared collateral are allocated to which positions

Project: Design choice.

Beyond: Acknowledged.

5.2.2 Missing Collateral Top-Up Function Prevents Users from Avoiding Liquidation

Description: When a borrower's collateral value falls below the required LTV ratio, they should be able to add more collateral to their existing position to prevent liquidation (which is a feature that should be added to the protocol in the near future). This is particularly problematic since the protocol does not provide any mechanism for borrowers to add more collateral to prevent liquidation.

Currently, the only options available to borrowers are:

1. Repay the full loan amount through `repay_debt()`
2. Get liquidated if they cannot repay in time

This creates unnecessary friction and losses for borrowers who may have the means to maintain a healthy collateral ratio but lack the mechanism to do so. The impact is especially severe during periods of high market volatility when collateral values can rapidly decrease.

The root cause is in the protocol's design which only allows collateral to be set at loan origination through `make_borrowing_offer_deposit()` and `make_borrowing_offer_allowance()`, with no subsequent modifications possible.

Recommendation: Implement a new `add_collateral()` function that allows borrowers to deposit additional collateral to their existing positions. The function should:

1. Accept parameters for the loan ID and additional collateral amount
2. Verify the caller is the borrower of the specified position
3. Transfer the additional collateral tokens from the borrower
4. Update the position's collateral value and LTV ratio
5. Emit an event for tracking

Project: Design choice.

Beyond: Acknowledged.

5.3 Low Risk

5.3.1 Incorrect decimal handling in lending offers can lead to value discrepancies

Description: The `make_lending_offer()` function assumes that all input amounts are in 18 decimals, but this assumption breaks when integrating with oracles in future updates. The function takes an `amount` parameter that represents the lending amount but does not perform any decimal normalization, which will lead to wrong value when using an oracle.

This can lead to value discrepancies when:

1. The oracle provides price feeds in different decimal places (e.g., 8 decimals for BTC/USD)
2. The underlying tokens have different decimal places (e.g., USDC with 6 decimals vs ETH with 18 decimals)

For example, if a user wants to lend 1000 USDC (6 decimals), they would pass in 10000000000000000000 (1000 * 10^18) but this would be interpreted as 1000000000 USDC (1000000000 * 10^6) - resulting in a massive overvaluation of the lending position.

The protocol already has utility functions `scale_to_18_decimals()` and `inverse_scale_to_18_decimals()` that handle decimal normalization, but they are not being used in this critical function.

Recommendation: All amounts should be normalized to 18 decimals when entering the system and denormalized when exiting. Update the function to use the existing decimal handling utilities:

```
- fn make_lending_offer(ref self: ContractState, token: ContractAddress, amount: u256,
↪  accepted_collateral: u256, price: Price) {
+ fn make_lending_offer(ref self: ContractState, token: ContractAddress, raw_amount: u256,
↪  accepted_collateral: u256, price: Price) {
+     let amount = scale_to_18_decimals(token, raw_amount);
      // ... rest of function
```

While the current implementation works as intended with manual decimal handling through the `inverse_scale()` function during `match_offer`, it would be preferable to implement this conversion directly in the code to avoid human error and facilitate future oracle integration (which use variable decimals). This improvement, however, is not critical at the moment and can be considered for future protocol updates.

Project: Acknowledged.

Beyond: Acknowledged.

5.3.2 Missing Nimbora Token Support in Asset Validation

Description: The `assert_is_lending_asset()` function performs validation of supported lending assets by checking if the provided token address matches a hardcoded list of allowed tokens. However, the function fails to include support for Nimbora protocol tokens that are mentioned in the constants file (e.g., `NIMBORA_nstUSD_ADDRESS`).

This oversight means that valid Nimbora tokens will be rejected by the validation check, preventing legitimate lending offers from being created with these assets. The impact is a reduction in protocol functionality and poor integration with the Nimbora ecosystem, as users attempting to create lending offers with supported Nimbora tokens will have their transactions revert unnecessarily.

The root cause is in the asset validation logic where the assertion only checks for basic tokens (ETH, USDT, USDC, DAI, DAIV0) while omitting Nimbora-specific tokens that are defined in the constants and intended to be supported by the protocol.

Recommendation: Update the asset validation check to include all supported Nimbora tokens. This can be done by adding the additional token addresses to the assertion check:

```
assert!(
    address == constants::ETH_ADDRESS ||
    address == constants::USDT_ADDRESS ||
    address == constants::USDC_ADDRESS ||
    address == constants::DAI_ADDRESS ||
    address == constants::DAIV0_ADDRESS ||
    address == constants::NIMBORA_nstUSD_ADDRESS ||
    address == constants::NIMBORA_nstUSD_ADDRESS,  // Add Nimbora tokens
    "Lending offer of only eth usdt usdc dai dai0 and Nimbora tokens are supported atm"
);
```

Alternatively, consider implementing a more maintainable solution using a whitelist mapping that can be updated by governance, rather than relying on hardcoded addresses in an assertion statement. This would make it easier to add or remove supported tokens in the future.

Project: Design choice.

Beyond: Acknowledged.

5.3.3 Code Organization Can Be Improved by Consolidating Stablecoin Category Checks

Description: The `category_id_from_address()` function contains separate conditional checks for stablecoins that return the same category ID (`USDC_CATEGORY`). Specifically, there is a standalone check for Nimbora stablecoin addresses (`NIMBORA_nsDAI_ADDRESS` and `NIMBORA_nstUSD_ADDRESS`) that is separated from the main stablecoin check block that handles `USDC_ADDRESS`, `DAI_ADDRESS`, and `DAIV0_ADDRESS`.

This separation of logically related checks:

1. Reduces code readability by splitting related logic
2. Makes maintenance more difficult as new stablecoin additions require checking multiple places
3. Increases the chance of inconsistencies when updating stablecoin handling
4. Makes the code more verbose without adding any functional benefit

While this doesn't present any security risks, it represents a departure from coding best practices that emphasize grouping related logic together.

Recommendation: Consolidate all stablecoin address checks into a single conditional block. This makes the code more maintainable and easier to understand.

```
- } else if address == constants::USDC_ADDRESS || address == constants::DAI_ADDRESS || address ==
↪  constants::DAIVO_ADDRESS {
+ } else if address == constants::USDC_ADDRESS ||
+          address == constants::DAI_ADDRESS ||
+          address == constants::DAIVO_ADDRESS ||
+          address == constants::NIMBORA_nsDAI_ADDRESS ||
+          address == constants::NIMBORA_nstUSD_ADDRESS {
    return constants::USDC_CATEGORY;
- }
- // ... other checks ...
- else if address == constants::NIMBORA_nsDAI_ADDRESS || address == constants::NIMBORA_nstUSD_ADDRESS {
-     return constants::USDC_CATEGORY;
}
```

Project: Design choice.

Beyond: Acknowledged.

5.3.4 Hardcoded Admin Address Reduces Protocol Flexibility and Security

Description: The protocol currently uses a hardcoded admin address (`ADMIN_ADDRESS`) as a constant in the contract. This approach creates several significant issues:

1. **Immutability Risk**: If the private key for this address is compromised, there is no way to transfer admin rights to a new address, leaving the protocol vulnerable or unusable.
2. **Deployment Inflexibility**: The hardcoded address requires code modifications and redeployment for different environments (testnet, mainnet) or when transferring ownership, increasing operational complexity and risk.
3. **Limited Access Control**: A simple hardcoded address lacks the sophisticated access control features available in established patterns like OpenZeppelin's Ownable, making it harder to implement proper governance or multi-signature controls.
4. **Development Constraints**: Without upgradeability during the development phase, any protocol changes require full redeployment, which can be costly and risky, especially when dealing with existing user funds or positions.

Recommendation: Implement the following improvements:

1. Replace the hardcoded admin address with OpenZeppelin's Ownable component or a similar ownership management system. This provides:
    - Secure ownership transfer functionality
    - Standard interfaces that other protocols recognize
    - Well-tested and audited access control

```
use openzeppelin::access::ownable::ownable::OwnableComponent::InternalTrait as OwnableInternalTrait;
use openzeppelin::upgrades::upgradeable::UpgradeableComponent::InternalTrait as
↪  UpgradeableInternalTrait;

component!(path: OwnableComponent, storage: ownable, event: OwnableComponentEvent);

#[abi(embed_v0)]
impl OwnableImpl = OwnableComponent::OwnableImpl<ContractState>;

#[storage]
    struct Storage {
        #[substorage(v0)]
        ownable: OwnableComponent::Storage,
    }
```

ref :

- OpenZeppelin components
- Starknet Cairo Ownable Component

This provides a more robust and flexible solution for protocol administration while maintaining security.

Project: Design choice.

Beyond: Acknowledged.

5.3.5 ERC20 Interface Compatibility Risk Due to Custom Implementation

Description: The protocol currently uses a custom ERC20 interface implementation rather than leveraging OpenZeppelin's battle-tested contracts. This creates compatibility issues with ERC20 tokens that may only implement one casing style for function names (snake_case vs camelCase).

While the tokens currently used in the protocol support both casing styles, this is not guaranteed for all ERC20 tokens that may be added in the future.

The root cause is in the `IERC20Dispatcher` interface implementation which assumes a specific function naming pattern. If a token only implements one casing style, function calls using the incompatible style will revert, breaking core protocol functionality like collateral transfers or loan repayments.

Recommendation: Replace the custom ERC20 interface with OpenZeppelin's standard implementation (that is referenced by the SNIP2 of Starknet: `https://github.com/OpenZeppelin/cairo-contracts/blob/main/packages/token/src/erc20/interface.cairo`) to ensure maximum compatibility with all ERC20 tokens. OpenZeppelin's contracts are extensively tested and handle edge cases around function naming conventions.

This change will standardize the ERC20 interface usage across the protocol and prevent integration issues with future token additions.

Project: Acknowledged.

Beyond: Acknowledged.