Morpho pre-liquidation Security Review

Auditors

Jonah Wu, Lead Security Researcher

Saw-mon and Natalie, Lead Security Researcher

Report prepared by: Lucas Goiriz

October 29, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

The Morpho Protocol is a decentralized, noncustodial lending protocol implemented for the Ethereum Virtual Machine

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Morpho preliquidation according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 3 days days in total, Morpho engaged with Spearbit to review the morpho-pre-liquidation protocol. In this period of time a total of **4** issues were found.

Summary

Project Name	Morpho
Repository	morpho-pre-liquidation
Commit	0ede4af7
Type of Project	DeFi, Lending
Audit Timeline	Oct 14th to Oct 17th
Fix period	Oct 18th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	0	0	0
Gas Optimizations	1	1	0
Informational	3	0	3
Total	**4**	**1**	**3**

5 Findings

5.1 Gas Optimization

5.1.1 The `LTV`-dependant ratio calculation in `preLIF` and `preLCF` can be refactor

Severity: Gas Optimization

Context: PreLiquidation.sol#L153, PreLiquidation.sol#L168

Description: The `LTV`-dependant ratio calculation in `preLIF` and `preLCF` can be refactored:

```
(ltv - PRE_LLTV).wDivDown(LLTV - PRE_LLTV)
```

Recommendation: Apply the following patch:

```
diff --git a/src/PreLiquidation.sol b/src/PreLiquidation.sol
index d4e91b2..c56208d 100644
--- a/src/PreLiquidation.sol
+++ b/src/PreLiquidation.sol
@@ -150,7 +150,8 @@ contract PreLiquidation is IPreLiquidation, IMorphoRepayCallback {
         require(borrowed > collateralQuoted.wMulDown(PRE_LLTV),
↪  ErrorsLib.NotPreLiquidatablePosition());

         uint256 ltv = borrowed.wDivUp(collateralQuoted);
-        uint256 preLIF = (ltv - PRE_LLTV).wDivDown(LLTV - PRE_LLTV).wMulDown(PRE_LIF_2 - PRE_LIF_1) +
↪  PRE_LIF_1;
+        uint256 factor = (ltv - PRE_LLTV).wDivDown(LLTV - PRE_LLTV);
+        uint256 preLIF = factor.wMulDown(PRE_LIF_2 - PRE_LIF_1) + PRE_LIF_1;

         if (seizedAssets > 0) {
             uint256 seizedAssetsQuoted = seizedAssets.mulDivUp(collateralPrice, ORACLE_PRICE_SCALE);
@@ -165,7 +166,7 @@ contract PreLiquidation is IPreLiquidation, IMorphoRepayCallback {

         // Note that the pre-liquidation close factor can be greater than WAD (100%).
         // In this case the position can be fully pre-liquidated.
-        uint256 preLCF = (ltv - PRE_LLTV).wDivDown(LLTV - PRE_LLTV).wMulDown(PRE_LCF_2 - PRE_LCF_1) +
↪  PRE_LCF_1;
+        uint256 preLCF = factor.wMulDown(PRE_LCF_2 - PRE_LCF_1) + PRE_LCF_1;

         uint256 repayableShares = uint256(position.borrowShares).wMulDown(preLCF);
         require(repaidShares <= repayableShares, ErrorsLib.PreLiquidationTooLarge(repaidShares,
↪  repayableShares));
```

`forge s --diff`:

```
testRedundantPreLiquidation((uint256,uint256,uint256,uint256,uint256,address)) (gas: -1977 (-0.000%))
testLCFDecreasing((uint256,uint256,uint256,uint256,uint256,address)) (gas: -78 (-0.090%))
testPreLIFDecreasing((uint256,uint256,uint256,uint256,uint256,address)) (gas: -78 (-0.090%))
testHighPreLltv((uint256,uint256,uint256,uint256,uint256,address)) (gas: -78 (-0.092%))
testLCFHigh((uint256,uint256,uint256,uint256,uint256,address)) (gas: -78 (-0.092%))
testHighPreLIF((uint256,uint256,uint256,uint256,uint256,address)) (gas: -78 (-0.093%))
testFactoryAddressZero() (gas: -35 (-0.093%))
testLowPreLIF((uint256,uint256,uint256,uint256,uint256,address)) (gas: -78 (-0.094%))
testNonexistentMarket((uint256,uint256,uint256,uint256,uint256,address)) (gas: -78 (-0.139%))
testOracle((uint256,uint256,uint256,uint256,uint256,address),uint256,uint256) (gas: -31104 (-1.633%))
testPreLiquidationWithInterest((uint256,uint256,uint256,uint256,uint256,address),uint256) (gas: -31366
↪  (-1.637%))
testPreLiquidationCallback((uint256,uint256,uint256,uint256,uint256,address),uint256,uint256) (gas:
↪  -31390 (-1.652%))
testPreLiquidationAssets((uint256,uint256,uint256,uint256,uint256,address),uint256,uint256) (gas:
↪  -31390 (-1.656%))
testPreLiquidationShares((uint256,uint256,uint256,uint256,uint256,address),uint256,uint256) (gas:
↪  -31390 (-1.659%))
testPreLiquidationLiquidatable((uint256,uint256,uint256,uint256,uint256,address),uint256,uint256,uint25
↪  6) (gas: -31104
↪  (-1.713%))
testNotPreLiquidatable((uint256,uint256,uint256,uint256,uint256,address),uint256,uint256) (gas: -31104
↪  (-1.727%))
testPreLiquidationTooLargeWithAssets((uint256,uint256,uint256,uint256,uint256,address),uint256,uint256,
↪  uint256) (gas: -31390
↪  (-1.729%))
testPreLiquidationTooLargeWithShares((uint256,uint256,uint256,uint256,uint256,address),uint256,uint256,
↪  uint256) (gas: -31390
↪  (-1.734%))
testCreatePreLiquidation((uint256,uint256,uint256,uint256,uint256,address)) (gas: -63600 (-1.955%))
testCreate2Deployment((uint256,uint256,uint256,uint256,uint256,address)) (gas: -64086 (-1.965%))
testInconsistentInput((uint256,uint256,uint256,uint256,uint256,address),uint256,uint256) (gas: -31131
↪  (-2.135%))
testEmptyPreLiquidation((uint256,uint256,uint256,uint256,uint256,address)) (gas: -31131 (-2.138%))
testNotMorpho((uint256,uint256,uint256,uint256,uint256,address)) (gas: -31128 (-2.140%))
Overall gas change: -505262 (-0.047%)
```

Morpho: Fixed in PR 85.

Spearbit: Fixed.

5.2 Informational

5.2.1 Iterative pre-liquidation

Severity: Informational

Context: PreLiquidation.sol#L90-L91, PreLiquidation.sol#L147-L150, PreLiquidation.sol#L166-L171

Description: Due to the following constraints:

```
require(WAD <= _preLiquidationParams.preLIF1, ErrorsLib.PreLIFTooLow());
require(_preLiquidationParams.preLIF1 <= _preLiquidationParams.preLIF2, ErrorsLib.PreLIFDecreasing());
```

$$10^{18} \leq \text{LIF}_1 \leq \text{LIF}_2$$

Not considering the division errors and gas costs (compared to the extracted value), it is guaranteed that the pre-liquidator always can extract some value.

The portion of borrowed assets where the pre-liquidator can pre-liquidate is limited due to the following `require` statement:

```
// Note that the pre-liquidation close factor can be greater than WAD (100%).
// In this case the position can be fully pre-liquidated.
uint256 preLCF = (ltv - PRE_LLTV).wDivDown(LLTV - PRE_LLTV).wMulDown(PRE_LCF_2 - PRE_LCF_1) + PRE_LCF_1;

uint256 repayableShares = uint256(position.borrowShares).wMulDown(preLCF);
require(repaidShares <= repayableShares, ErrorsLib.PreLiquidationTooLarge(repaidShares,
↪  repayableShares));
```

One can preform the pre-liquidation if:

$$\text{LLTV}_{pre} < \text{LTV} \leq \text{LLTV}$$

For certain parameters there might be cases that after pre-liquidation the LTV stays in the above range (even though it might have been slightly improved) and thus the pre-liquidator can keep pre-liquidating as long as LTV stays in the above range which might potentially allow the pre-liquidator to pre-liquidate most of the borrowed assets.

Recommendation: The above scenario needs to be analysed.

Morpho: Acknowledged. A warning regarding this issue has been added to `README.md` in PR 85.

Spearbit: Acknowledged.

5.2.2 Different `SafeTransferLib` has been used compared to `morpho-blue`

Severity: Informational

Context: PreLiquidation.sol#L12

Description: In `morpho-blue` a custom `SafeTransferLib` has been used where as in the project one is using the library from `solmate`. Besides one being written in assembly and the other in a higher-level solidity language. The main difference is that the custom `SafeTransferLib` has the extra check:

```
require(address(token).code.length > 0, ErrorsLib.NO_CODE);
```

Which was added in PR629: fix(safe-transfer-lib): check for code

Recommendation: It would be great to keep using the same library. But it is not necessary since the check above has already been performed when a user borrows a loan in `borrow` using the custom library. And so for a position with a non-zero `position.borrowShares` it has already been checked that `LOAN_TOKEN` has a non-zero code length.

It should at least be noted/documented.

Morpho: Acknowledged. The reason why the morpho-blue library hasn't been used is that it doesn't contain `safeApprove`, and the pre-liquidation contract needs to approve Morpho. In order to use morpho-blue's one, we could either:

1. Have a second lib with just `safeApprove`.
2. Use solmate for `safeApprove`.
3. Don't `safeApprove` and just `approve` (this should be fine too).

Spearbit: Acknowledged.

5.2.3 Solidity version

Severity: Informational

Context: Global scope

Description: The solidity versions used for some of the contracts is `solc 0.8.27`. This version includes new opcodes/bytecodes that depending on the `evm_version` selected can introduce `PUSH0`, `MCOPY`,...

Recommendation: Make sure the `evm_version` used with all the new opcodes are supported for the corresponding chains that the contracts are going to be deployed on.

Morpho: Acknowledged. After compiling the `PreLiquidation` and `PreLiquidationFactory` contracts, no new opcodes were found (checked `PUSH0`, `MCOPY`, `TLOAD` and `TSTORE`) though they might appear with through another compiler configuration.

Spearbit: Acknowledged.