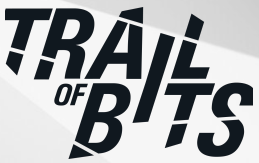

Morpho Protocol V1

Security Assessment

July 21, 2022

Prepared for:
Paul Frambot and Merlin Egalite
Morpho Labs

Prepared by: **Michael Colburn, Felipe Manzano, and Bo Henderson**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Morpho Labs engaged Trail of Bits to review the security of its Morpho-Compound smart contracts. From May 9 to May 20, 2022, a team of three consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and documentation. We performed static analysis and a manual review of the project's Solidity components.

Summary of Findings

The audit uncovered significant flaws that could impact system integrity or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Low	1
Informational	3
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	2
Auditing and Logging	1
Data Validation	4
Timing	1

Notable Findings

A significant flaw that impacts system integrity or availability is listed below.

- **TOB-MORPHO-6**
  The main `Morpho` contract will be deployed behind a `delegatecall`-based proxy, but it also uses `delegatecall` itself to enable a more modular architecture, circumventing contract size limits. An attacker is able to bypass the proxy and initialize the contract to trigger a call to `selfdestruct`, which would halt the system.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineers were associated with this project:

Michael Colburn, Consultant
michael.colburn@trailofbits.com

Felipe Manzano, Consultant
felipe.manzano@trailofbits.com

Bo Henderson, Consultant
bo.henderson@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
May 5, 2022	Pre-project kickoff call
May 15, 2022	Status update meeting #1
May 20, 2022	Delivery of report draft
May 24, 2022	Report readout meeting
June 4, 2022	Delivery of final report
July 21, 2022	Addition of appendix F: Additional Morpho Team Findings

Project Goals

The engagement was scoped to provide a security assessment of the Morpho-Compound smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are there arithmetic rounding issues that affect the code?
- Is it possible to steal funds or lose tokens?
- Could participants manipulate the contracts in unexpected ways?
- Are appropriate access controls in place for user and controller roles?
- Could different pools deviate in the way they operate?
- Could the underlying Morpho account in Compound be liquidated?

Project Targets

The engagement involved a review and testing of the following target.

`morpho-contracts`

Repository	https://github.com/morpho-labs/morpho-contracts
Version	`8bdee7ccf4e20902b77290e672f848360525d9a4`
Type	Solidity
Platform	EVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. We used Slither to scan for common vulnerabilities and performed a manual review over the following contracts:

- **`Morpho` and the contracts it inherits from**. `Morpho` is the main contract of the protocol and the entrypoint through which users supply and borrow liquidity. This contract inherits from the `MorphoGovernance`, `MorphoUtils`, and `MorphoStorage` contracts, among others, which all define helper and administrative functions to manage the liquidity pool. We reviewed `Morpho` and the contracts it inherits from to ensure the soundness of the system's architecture and to detect issues related to the use of `delegatecall`.

- **`PositionsManager`**. This contract inherits from the `MatchingEngine` and `MorphoStorage` contracts, which provide the core functionality for managing users' deposits, withdrawals, borrowing operations, and repayments. The `PositionsManager` contract acts as a library that the main `Morpho` entrypoint contract calls (via `delegatecall`). We reviewed `PositionsManager` to ensure that it performs accurate bookkeeping and properly liquidates users at the Morpho level. We also searched for edge cases that could cause the underlying Morpho account in Compound to be liquidated.

- **`InterestRatesManager`**. This contract contains the arithmetic required to calculate accrued interest. Like `PositionsManager`, it inherits from `MorphoStorage` and acts as a library that the main `Morpho` entrypoint contract calls (via `delegatecall`). We reviewed `InterestRatesManager` to ensure that it performs accurate bookkeeping for peer-to-peer (P2P) rates.

- **`RewardsManager`**. This contract contains the logic required to manage and claim liquidity mining rewards provided by the underlying Compound protocol. It manages its own local storage and is called by the main `Morpho` contract to pass rewards to users. We reviewed this contract to ensure that it accurately tracks users' rewards and stays in sync with Compound.

- **`IncentivesVault`**. This contract contains the logic required to manage protocol fees collected by the Morpho DAO. It manages its own local storage and is called by the main `Morpho` contract. It also gives users the option to claim rewards in Morpho tokens rather than COMP tokens. We reviewed `IncentivesVault` to ensure that it properly calculates incentive amounts for users.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- The `Lens` contract and the contracts in the `aave` folder were explicitly out of scope for this review.
- The code for the Morpho token was not reviewed, as it was not yet available at the time of the audit.
- We did not use fuzz testing to perform dynamic analysis of the system.
- The user interface and other off-chain components were not reviewed.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The codebase uses a version of the Solidity compiler that includes built-in overflow detection. Checks are in place for certain edge cases that may be introduced by rounding errors, and the Morpho team is investigating alternative approaches to implementing arithmetic to reduce the impact of these rounding errors.	**Moderate**
Auditing	We identified only one issue related to auditing: certain events emitted by the system do not provide complete information about the corresponding state changes (TOB-MORPHO-2). The Morpho team is actively developing an incident response plan to address a wide variety of scenarios.	**Satisfactory**
Authentication / Access Controls	While we identified a function missing an access control check that would provide additional defense-in-depth protection (TOB-MORPHO-3), the system generally has sufficient access controls in place.	**Satisfactory**
Complexity Management	The codebase under review was stable for the duration of the assessment. The specific responsibilities for each of the contracts are clearly defined, and functions are scoped appropriately.	**Satisfactory**
Decentralization	The Morpho team plans to use Zodiac to implement decentralized governance. The various governance-controlled operations and multisignature thresholds are sufficiently documented.	**Satisfactory**

Documentation	The Morpho yellow paper and the codebase's inline comments provide extensive documentation. However, some comments contain typos or appear to have been copied and pasted incorrectly, and several sections of the yellow paper are not yet complete.	**Satisfactory**
Front-Running Resistance	Further investigation into potential front-running vulnerabilities in the P2P wait list is needed.	**Further Investigation Required**
Low-Level Manipulation	In using a proxy pattern, the codebase calls assembly and low-level code, but only minimally.	**Satisfactory**
Testing and Verification	The project's test suite has good coverage, is integrated in the continuous integration process, and includes fuzz tests. However, the unit tests related to small threshold values and dust management could be improved to explore more scenarios.	**Satisfactory**
Upgradeability	The tests and documentation focused on upgrading contracts is insufficient. We discovered a high-severity issue related to contract upgradeability (TOB-MORPHO-6), and we suspect that others may be present.	**Weak**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Lack of two-step process for contract ownership changes	Data Validation	High
2	Incomplete information provided in Withdrawn and Repaid events	Auditing and Logging	Informational
3	Missing access control check in withdrawLogic	Access Controls	Informational
4	Lack of zero address checks in setter functions	Data Validation	Informational
5	Risky use of toggle functions	Timing	Low
6	Anyone can destroy Morpho's implementation	Access Controls	High
7	Lack of return value checks during token transfers	Data Validation	Undetermined
8	Risk of loss of precision in division operations	Data Validation	Undetermined

Detailed Findings

1. Lack of two-step process for contract ownership changes	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MORPHO-1
Target: Throughout the codebase	

Description

The owner of the `IncentivesVault` contract and other `Ownable` Morpho contracts can be changed by calling the `transferOwnership` function. This function internally calls the `_transferOwnership` function, which immediately sets the contract's new owner. Making such a critical change in a single step is error-prone and can lead to irrevocable mistakes.

```
13  contract IncentivesVault is IIncentivesVault, Ownable {
```

Figure 1.1: Inheritance of `contracts/compound/IncentivesVault.sol`

```
62  function transferOwnership(address newOwner) public virtual onlyOwner {
63      require(newOwner != address(0), "Ownable: new owner is the zero address");
64      _transferOwnership(newOwner);
65  }
```

Figure 1.2: The `transferOwnership` function in `@openzeppelin/contracts/access/Ownable.sol`

Exploit Scenario

Bob, the `IncentivesVault` owner, invokes `transferOwnership()` to change the contract's owner but accidentally enters the wrong address. As a result, he permanently loses access to the contract.

Recommendations

Short term, for contract ownership transfers, implement a two-step process, in which the owner proposes a new address and the transfer is completed once the new address has executed a call to accept the role.

Long term, identify and document all possible actions that can be taken by privileged accounts and their associated risks. This will facilitate reviews of the codebase and prevent future mistakes.

2. Incomplete information provided in Withdrawn and Repaid events

Severity: **Informational**	Difficulty: **High**
Type: Auditing and Logging	Finding ID: TOB-MORPHO-2
Target: `contracts/compound/PositionsManager.sol`	

Description

The core operations in the `PositionsManager` contract emit events with parameters that provide information about the operations' actions. However, two events, `Withdrawn` and `Repaid`, do not provide complete information.

For example, the `withdrawLogic` function, which performs withdrawals, takes a `_supplier` address (the user supplying the tokens) and `_receiver` address (the user receiving the tokens):

```
    /// @param _supplier The address of the supplier.
    /// @param _receiver The address of the user who will receive the tokens.
    /// @param _maxGasForMatching The maximum amount of gas to consume within a
matching engine loop.
    function withdrawLogic(
        address _poolTokenAddress,
        uint256 _amount,
        address _supplier,
        address _receiver,
        uint256 _maxGasForMatching
    ) external
```

Figure 2.1: The function signature of `PositionsManager`'s `withdrawLogic` function

However, the corresponding event in `_safeWithdrawLogic` records only the `msg.sender` of the transaction, so the `_supplier` and `_receiver` involved in the transaction are unclear. Moreover, if a withdrawal is performed as part of a liquidation operation, three separate addresses may be involved—the `_supplier`, the `_receiver`, and the `_user` who triggered the liquidation—and those monitoring events will have to cross-reference multiple events to understand whose tokens moved where.

```
    /// @notice Emitted when a withdrawal happens.
    /// @param _user The address of the withdrawer.
    /// @param _poolTokenAddress The address of the market from where assets are
withdrawn.
    /// @param _amount The amount of assets withdrawn (in underlying).
    /// @param _balanceOnPool The supply balance on pool after update.
```

```
/// @param _balanceInP2P The supply balance in peer-to-peer after update.
event Withdrawn(
    address indexed _user,
    …
```

Figure 2.2: The declaration of the `Withdrawn` event in `PositionsManager`

```
        emit Withdrawn(
            msg.sender,
            _poolTokenAddress,
            _amount,
            supplyBalanceInOf[_poolTokenAddress][msg.sender].onPool,
            supplyBalanceInOf[_poolTokenAddress][msg.sender].inP2P
        );
```

Figure 2.3: The emission of the `Withdrawn` event in the `_safeWithdrawLogic` function

A similar issue is present in the `_safeRepayLogic` function's `Repaid` event.

Recommendations

Short term, add the relevant addresses to the `Withdrawn` and `Repaid` events.

Long term, review all of the events emitted by the system to ensure that they emit sufficient information.

3. Missing access control check in withdrawLogic

Severity: **Informational**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-MORPHO-3
Target: `contracts/compound/PositionsManager.sol`	

Description

The `PositionsManager` contract's `withdrawLogic` function does not perform any access control checks. In practice, this issue is not exploitable, as all interactions with this contract will be through `delegatecall`s with a hard-coded `msg.sender` sent from the main `Morpho` contract. However, if this code is ever reused or if the architecture of the system is ever modified, this guarantee may no longer hold, and users without the proper access may be able to withdraw funds.

```
    /// @dev Implements withdraw logic with security checks.
    /// @param _poolTokenAddress The address of the market the user wants to
interact with.
    /// @param _amount The amount of token (in underlying).
    /// @param _supplier The address of the supplier.
    /// @param _receiver The address of the user who will receive the tokens.
    /// @param _maxGasForMatching The maximum amount of gas to consume within a
matching engine loop.
    function withdrawLogic(
        address _poolTokenAddress,
        uint256 _amount,
        address _supplier,
        address _receiver,
        uint256 _maxGasForMatching
    ) external {
```

Figure 3.1: The `withdrawLogic` function, which takes a supplier and whose comments note that it performs security checks

Recommendations

Short term, add a check to the `withdrawLogic` function to ensure that it withdraws funds only from the `msg.sender`.

Long term, implement security checks consistently throughout the codebase.

4. Lack of zero address checks in setter functions

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MORPHO-4
Target: `contracts/compound/MorphoGovernance.sol`	

Description

Certain setter functions fail to validate incoming arguments, so callers can accidentally set important state variables to the zero address. A mistake like this could initially go unnoticed because a `delegatecall` to an address without code will return success.

```
/// @notice Sets the `positionsManager`.
/// @param _positionsManager The new `positionsManager`.
function setPositionsManager(IPositionsManager _positionsManager) external onlyOwner
{
    positionsManager = _positionsManager;
    emit PositionsManagerSet(address(_positionsManager));
}
```

Figure 4.1: An important address setter in `MorphoGovernance`

Exploit Scenario

Alice and Bob control a multisignature wallet that is the owner of a deployed `Morpho` contract. They decide to set `_positionsManager` to a newly upgraded contract but, while invoking `setPositionsManager`, they mistakenly omit the address. As a result, `_positionsManager` is set to the zero address, resulting in undefined behavior.

Recommendations

Short term, add zero-value checks to all important address setters to ensure that owners cannot accidentally set addresses to incorrect values, misconfiguring the system. Specifically, add zero-value checks to the `setPositionsManager`, `setRewardsManager`, `setInterestRates`, `setTreasuryVault`, and `setIncentivesVault` functions, as well as the `_cETH` and `_cWeth` parameters of the `initialize` function in the `MorphoGovernance` contract.

Long term, incorporate Slither into a continuous integration pipeline, which will continuously warn developers when functions do not have checks for zero values.

5. Risky use of toggle functions	
Severity: **Low**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-MORPHO-5
Target: `contracts/compound/MorphoGovernance.sol`	

Description

The codebase uses a toggle function, `togglePauseStatus`, to pause and unpause a market. This function is error-prone because setting a pause status on a market depends on the market's current state. Multiple uncoordinated pauses could result in a failure to pause a market in the event of an incident.

```
/// @notice Toggles the pause status on a specific market in case of emergency.
/// @param _poolTokenAddress The address of the market to pause/unpause.
function togglePauseStatus(address _poolTokenAddress)
    external
    onlyOwner
    isMarketCreated(_poolTokenAddress)
{
    Types.MarketStatus storage marketStatus_ = marketStatus[_poolTokenAddress];
    bool newPauseStatus = !marketStatus_.isPaused;
    marketStatus_.isPaused = newPauseStatus;
    emit PauseStatusChanged(_poolTokenAddress, newPauseStatus);
}
```

Figure 5.1: The `togglePauseStatus` method in `MorphoGovernance`

This issue also applies to `togglePartialPauseStatus`, `toggleP2P`, and `toggleCompRewardsActivation` in `MorphoGovernance` and to `togglePauseStatus` in `IncentivesVault`.

Exploit Scenario

All signers of a 4-of-9 multisignature wallet that owns a `Morpho` contract notice an ongoing attack that is draining user funds from the protocol. Two groups of four signers hurry to independently call `togglePauseStatus`, resulting in a failure to pause the system and leading to the further loss of funds.

Recommendations

Short term, replace the toggle functions with ones that explicitly set the pause status to `true` or `false`.

Long term, carefully review the incident response plan and ensure that it leaves as little room for mistakes as possible.

6. Anyone can destroy Morpho's implementation

Severity: **High**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-MORPHO-6
Target: `contracts/compound/Morpho*.sol`	

Description

An incorrect access control on the `initialize` function for Morpho's implementation contract allows anyone to destroy the contract.

Morpho uses the `delegatecall` proxy pattern for upgradeability:

```
abstract contract MorphoStorage is OwnableUpgradeable, ReentrancyGuardUpgradeable {
```

Figure 6.1: contracts/compound/MorphoStorage.sol#L16

With this pattern, a proxy contract is deployed and executes a `delegatecall` to the implementation contract for certain operations. Users are expected to interact with the system through this proxy. However, anyone can also directly call Morpho's implementation contract.

Despite the use of the proxy pattern, the implementation contract itself also has `delegatecall` capacities. For example, when called in the `updateP2PIndexes` function, `setReserveFactor` executes a `delegatecall` on user-provided addresses:

```
function setReserveFactor(address _poolTokenAddress, uint16 _newReserveFactor)
    external
    onlyOwner
    isMarketCreated(_poolTokenAddress)
{
    if (_newReserveFactor > MAX_BASIS_POINTS) revert ExceedsMaxBasisPoints();
    updateP2PIndexes(_poolTokenAddress);
```

Figure 6.2: contracts/compound/MorphoGovernance.sol#L203-L209

```
function updateP2PIndexes(address _poolTokenAddress) public {
    address(interestRatesManager).functionDelegateCall(
        abi.encodeWithSelector(
            interestRatesManager.updateP2PIndexes.selector,
            _poolTokenAddress
        )
    );
}
```

Figure 6.3: contracts/compound/MorphoUtils.sol#L119-L126

These functions are protected by the `onlyOwner` modifier; however, the system's owner is set by the `initialize` function, which is callable by anyone:

```
function initialize(
    IPositionsManager _positionsManager,
    IInterestRatesManager _interestRatesManager,
    IComptroller _comptroller,
    Types.MaxGasForMatching memory _defaultMaxGasForMatching,
    uint256 _dustThreshold,
    uint256 _maxSortedUsers,
    address _cEth,
    address _wEth
) external initializer {
    __ReentrancyGuard_init();
    __Ownable_init();
```

Figure 6.4: contracts/compound/MorphoGovernance.sol#L114-L125

As a result, anyone can call `Morpho.initialize` to become the owner of the implementation and execute any `delegatecall` from the implementation, including to a contract containing a `selfdestruct`.

Doing so will cause the proxy to point to a contract that has been destroyed.

This issue is also present in `PositionsManagerForAave`.

Exploit Scenario

The system is deployed. Eve calls `Morpho.initialize` on the implementation and then calls `setReserveFactor`, triggering a `delegatecall` to an attacker-controlled contract that self-destructs. As a result, the system stops working.

Recommendations

Short term, add a constructor in `MorphoStorage` and `PositionsManagerForAaveStorage` that will set an `is_implementation` variable to `true` and check that this variable is `false` before executing any critical operation (such as `initialize`, `delegatecall`, and `selfdestruct`). By setting this variable in the constructor, it will be set only in the implementation and not in the proxy.

Long term, carefully review the pitfalls of using the `delegatecall` proxy pattern.

References

- Breaking Aave Upgradeability

7. Lack of return value checks during token transfers	
Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-MORPHO-7
Target: `contracts/compound/IncentivesVault.sol`	

Description
In certain parts of the codebase, contracts that execute transfers of the Morpho token do not check the values returned from those transfers.

The development of the Morpho token was not yet complete at the time of the audit, so we were unable to review the code specific to the Morpho token. Some tokens that are not ERC20 compliant return `false` instead of reverting, so failure to check such return values could result in undefined behavior, including the loss of funds. If the Morpho token adheres to ERC20 standards, then this issue may not pose a risk; however, due to the lack of return value checks, the possibility of undefined behavior cannot be eliminated.

```
function transferMorphoTokensToDao(uint256 _amount) external onlyOwner {
    morphoToken.transfer(morphoDao, _amount);
    emit MorphoTokensTransferred(_amount);
}
```

Figure 7.1: The `transerMorphoTokensToDao` method in `IncentivesVault`

Exploit Scenario
The Morpho token code is completed and deployed alongside the other Morpho system components. It is implemented in such a way that it returns `false` instead of reverting when transfers fail, leading to undefined behavior.

Recommendations
Short term, consider using a `safeTransfer` library for all token transfers.

Long term, review the token integration checklist and check all the components of the system to ensure that they interact with tokens safely.

8. Risk of loss of precision in division operations	
Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-MORPHO-8
Target: `contracts/compound/PositionsManager.sol`	

Description
A common pattern in the codebase is to divide a user's debt by the total supply of a token; a loss of precision in these division operations could occur, which means that the supply delta would not account for the entire matched delta amount. The impact of this potential loss of precision requires further investigation.

For example, the `borrowLogic` method uses this pattern:

```
    toWithdraw += matchedDelta;
    remainingToBorrow -= matchedDelta;
    delta.p2pSupplyDelta -= matchedDelta.div(poolSupplyIndex);
    emit P2PSupplyDeltaUpdated(_poolTokenAddress, delta.p2pSupplyDelta);
```

Figure 8.1: Part of the `borrowLogic()` method

Here, if `matchedDelta` is not a multiple of `poolSupplyIndex`, the remainder would not be taken into account. In an extreme case, if `matchedDelta` is smaller than `poolSupplyIndex`, the result of the division operation would be zero.

An attacker could exploit this loss of precision to extract small amounts of underlying tokens sitting in the `Morpho` contract.

Exploit Scenario
Bob transfers some Dai to the `Morpho` contract by mistake. Eve sees this transfer, deposits some collateral, and then borrows an amount of Dai from `Morpho` small enough that it does not affect Eve's debt. Eve withdraws her deposited collateral and walks out with Bob's Dai. Further investigation into this exploit scenario is required.

Recommendations
Short term, add checks to validate input data to prevent precision issues in division operations.

Long term, review all the arithmetic that is vulnerable to rounding issues.

Summary of Recommendations

The Morpho Protocol is a work in progress with multiple planned iterations. Trail of Bits recommends that Morpho Labs address the findings detailed in this report and take the following additional steps prior to deployment:

- Continue to improve the system's documentation, including the yellow paper specification and the incident response plan.
- Continue to explore alternative approaches to implementing arithmetic that will reduce the number of rounding and loss-of-precision errors.
- Integrate static analysis tooling, such as Slither, into the continuous integration pipeline.
- Consider adding Echidna to the test suite to augment Foundry's fuzz testing (which is already implemented).
- Carefully review the pitfalls of using the `delegatecall` proxy pattern.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Upgradeability	Related to contract upgradeability

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- The following functions are defined but never used. Consider removing them to enhance code readability and clean up the codebase.
  - `DoubleLinkedList.getPrev`
  - `DoubleLinkedList.getTail`
  - `CompoundMath.average`
- Consider changing the state variable `_isLiquidable` to `_isLiquidatable`. The word "liquidatable" is more easily recognizable as meaning "able to be liquidated" than the word "liquidable."
- Some local variables are declared without being initialized and are then used in arithmetic operations that assume that their values start at zero. Although this does not introduce any bugs, this pattern relies on assumptions about the compiler and negatively impacts code readability. Consider having variables explicitly initialized to zero when they are declared.

D. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the below output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

E. Morpho Team Findings

During the assessment, the Morpho Labs team independently identified several issues in the codebase. We verified the existence of these issues, which are listed below to provide a complete report of the audit:

- If the peer-to-peer (P2P) lending market is disabled, the delta will no longer be updated, which could cause the underlying Morpho account on Compound to be liquidatable. This is a high-severity issue, but the fix is straightforward: the delta-updating operations should be moved out of the `p2pDisabled` conditional block. The Morpho Labs team is also investigating other options to fix this issue.
- The `updateP2PIndexes` function does not check whether a particular market has been created. For safety purposes, an `isMarketCreated` check should be added to `updateP2PIndexes`.
- When a market is created, its index cursor is not set. Code should be added to ensure the index cursor is set when a market is created.
- Some areas of the codebase are vulnerable to rounding errors. The Morpho Labs team is investigating other approaches to implementing arithmetic to eliminate this risk.
- Issues involving the rewards activation status could allow an attacker to drain COMP rewards from every user or even to drain the incentives vault.

F. Additional Morpho Team Findings

Following the assessment, the Morpho Labs team independently identified several additional issues in the codebase. From July 13 to July 15, 2022, a team of two engineers worked to verify the existence of these issues and Morpho Labs's fixes for them, which are listed below:

- Due to the permissionless relaying of oracle updates in Compound's Open Price Feeds, an attacker could front-run an oracle update and max out Morpho's borrowing capacity in Compound, forcing Morpho into a state in which it could be liquidated after the oracle update.

  To profitably exploit this issue would require precise timing and very large capital; therefore, the Morpho team removed the ability to borrow and repay in the same block, effectively blocking flash loans.

- Tokens donated to an underlying `cToken` contract could cause Morpho's accounting to become out of sync with the actual values in Compound. Because of this issue, an arithmetic overflow could occur, causing transactions to revert unexpectedly and resulting in a denial of service.

  The Morpho Labs team updated the `InterestRatesManager` logic to check for the potential overflow and update the internal bookkeeping to prevent it from occurring.

- A logic error in the calculation of updated reward indices could cause some transactions to revert, including matching operations involving affected users.

  The Morpho Labs team updated the logic to correctly calculate the reward indices.

- Morpho-Compound does not claim COMP rewards proactively for each asset, allowing rewards from one asset to be used for all eligible Morpho users. This optimization could temporarily prevent some users from claiming their rewards.

  The Morpho Labs team updated the rewards logic to always claim the underlying rewards, regardless of whether the system already has a sufficient balance to pay out.

- A logic error in the P2P borrowing rate calculation could result in slightly higher-than-necessary borrowing rates for users.

  The Morpho Labs team updated the `shareOfTheDelta` calculations to more accurately reflect the state of the system.

- An event emitted during repayment logs the P2P borrow delta instead of the supply delta.

  The Morpho Labs team updated it to emit the correct variable.

- The `delegatecall` library used by Morpho does not enforce contracts existence checks or check that targets were set to nonzero addresses. This is not an issue as it is currently used, but it could be in future versions of the codebase.

  This issue has not been fixed, but the Morpho Labs team has acknowledged it.

Additionally, we performed a high-level review of the other changes to the Morpho Compound codebase introduced since the previous assessment. We did not find any issues related to these changes.